NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

EVALUATION OF PROGRAM SPECIFICATION AND VERIFICATION SYSTEMS

by

Sonali Ubhayakar

June 2003

Thesis Advisor:	George Dinolt
Co-Advisor:	Tim Levin

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** June 2003	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE: Evalutation of Program Specification and Verification Systems	**5. FUNDING NUMBERS**
6. AUTHOR(S) Ubhayakar, Sonali S.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT

Computer systems that earn a high degree of trust must be backed by rigorous verification methods. A verification system is an interactive environment for writing formal specifications and checking formal proofs. Verification systems allow large complicated proofs to be managed and checked interactively. We desire evaluation criteria that provide a means of finding which verification system is suitable for a specific research environment and what needs of a particular project the tool satisfies. Therefore, the purpose of this thesis is to develop a methodology and set of evaluation criteria to evaluate verification systems for their suitability to improve the assurance that systems meet security objectives. A specific verification system is evaluated with respect to the defined methodology. The main goals are to evaluate whether the verification system has the capability to express the properties of software systems and to evaluate whether the verification system can provide inter-level mapping, a feature required for understanding how a system meets security objectives.

4. SUBJECT TERMS Verification System, PVS, ACL2, Formal Methods, Inter-level Mapping, Assurance, Verifiable Protection	**15. NUMBER OF PAGES** 159
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

THIS PAGE INTENTIONALLY LEFT BLANK

EVALUATION OF PROGRAM SPECIFICATION AND VERIFICATION SYSTEMS

Sonali S. Ubhayakar
Civilian, Naval Postgraduate School
B.S., University of California at Los Angeles, 2001

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE
from the

NAVAL POSTGRADUATE SCHOOL
June 2003

Author: Sonali S. Ubhayakar

Approved by: George Dinolt, Thesis Advisor

Timothy Levin, Co-Advisor

Peter Denning, Chair
Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Computer systems that earn a high degree of trust must be backed by rigorous verification methods. A verification system is an interactive environment for writing formal specifications and checking formal proofs. Verification systems allow large complicated proofs to be managed and checked interactively. We desire evaluation criteria that provide a means of finding which verification system is suitable for a specific research environment and what needs of a particular project the tool satisfies. Therefore, the purpose of this thesis is to develop a methodology and set of evaluation criteria to evaluate verification systems for their suitability to improve the assurance that systems meet security objectives. A specific verification system is evaluated with respect to the defined methodology. The main goals are to evaluate whether the verification system has the capability to express the properties of software systems and to evaluate whether the verification system can provide inter-level mapping, a feature required for understanding how a system meets security objectives.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my advisors, Dr. George Dinolt and Professor Timothy Levin, for all their help and understanding while working on this thesis. Without them, I do not think I would have learned and accomplished what I have in the last few months. I especially want to thank them for their patience and friendly attitudes in every meeting and discussion.

I would like to thank NSF for the SFS program here at the Naval Postgraduate School. Lastly, my advisors and I would like to acknowledge the support from the DARPA CHATS program under DARPA contract #R3W01 – R3W99.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

"Use theory to provide insight; use common sense and intutition where it is suitable, but fall back upon the formal theory when difficulties and complexities arise."[1]

Computer systems that earn a high degree of trust must be backed by rigorous verification methods. Formal methods deal with formally reasoning about computer systems and whether those systems offer verifiable protection. Without the use of verification systems, formal verification is impractical. These tools help to ensure the absence of subversion by automating the processes of generating proofs and evaluation evidence. Recognition is growing for the value of formal methods in certain area such as verifying algorithms and exploring properties of complex interactions. The purpose of this paper is to develop evaluation methodology for a set of verification systems that will be used to improve the assurance that systems meet security objectives.

This thesis is divided into three phases: the desktop analysis phase, the hands-on phase, and the evaluation phase. First, a list is developed of potential verification systems for evaluation. Then a desktop analysis is performed of the candidates where the potential verification systems will be analyzed to find out whether they meet a set of broad functional requirements (evaluation criteria). Defining evaluation criteria provides a means of finding which verification system is suitable for a specific research environment and what needs of a particular project the tool satisfies. Therefore, the selected verification system(s) are the ones that best satisfy the evaluation criteria.

A hands-on study or empirical study of the selected verification system is performed. First, a simplified version of a complex computer security related problem is constructed. Then a formally security policy and a formal top level specification is developed in terms of the selected verification system. From there, a mapping of the FTLS to the formal security policy model is attempted. The major goal of this paper is to

[1] From Gries, David. *The Science of Programming*. 1981, pages 164 - 165

illustrate whether the verification system chosen for evaluation can demonstrate inter-level mapping, a feature that is required in later applications.

The verification system that is chosen for analysis is evaluated based on a set of questions. These questions are: how well does the system describe the security policy and the FTLS, how much assurance is provided, and how easy is it to prove that the FTLS implements the policy? The main features of the verification system that are analyzed to answer how well the verification system described the FTLS and the formal security policy model are the system's expressiveness, its consistency checking mechanisms, and any features that help in improving the specifications. Whether the system provides mechanisms for checking semantic and logical consistency can help to determine the level of assurance provided. Lastly, decision and inference strategies of the system's theorem prover, the ease of use of the interface, and the implementation language of the theorem prover are the main features that can establish how easy it is to prove that the FTLS satisfies the policy. The main concern, however, is that the system must first demonstrate the ability to provide inter-level mapping before a thorough analysis can be constructed.

Our empirical study of the verification system that we chose to evaluate paves the way for future research of other verification systems. The methodology used in this thesis can be followed for the analysis of the next selected verification system. The inter-level mapping work that has been done in this paper demonstrates new research that has not been addressed in any PVS documentation that we have read. Therefore, the problems we have come upon as well as the ideas we have brought forth should be assessed and used in future work.

"If you wish to derive the most profit from your effort, look out for such features of a problem at hand as may be useful in handling the problems to come. A solution that you have obtained by your own effort or one that you have read or heard, but have followed with real interest and insight, may become a pattern for you, a model that you can imitate with advantage in solving similar problems...."[2]

[2] Polya, George. *Mathematical Discovery: On Understanding, Learning and Teaching Problem Solving*, Wiley, combined edition, 1981, preface

I. INTRODUCTION

A. PURPOSE

Computer systems that earn a high degree of trust must be backed by rigorous verification methods. Formal methods deal with formally reasoning about computer systems and whether those systems offer verifiable protection. "The IEEE definition of verification is, confirmation by examination and provisions of objective evidence that specified requirements have been fulfilled". [Asc02] Formal verification is impractical without the use of tools such as model-checkers and theorem provers. These tools help to ensure the absence of subversion by automating the processes of generating proofs and evaluation evidence. Recognition is growing for the value of formal methods in certain areas such as verifying algorithms and exploring properties of complex interactions. Tools are available that allow large complicated proofs to be managed and checked interactively. However, many in the computer science community agree that these verification systems may not make the activity of formal proof easy and may not be easy to use.

The main classes of verification tools include theorem proving (both automatic and human-assisted) and model checking approaches. The integrations of both these approached is advocated and exemplified in current verification tools. Tool support is important in formal methods, however it is a very skilled and time consuming activity. There are theorem provers that integrate inductive proof techniques, general rewrite procedures, model checking, propositional provers, linear arithmetic, other decision procedures, tactic facilities, execution capabilities, enhanced static checking, lemma generation, and computer algebra systems (See glossary). Some of the systems that are at least partially successful in combining these techniques are PVS, HOL, EVES, and ACL2. Another development direction of importance is the automation of verification procedures most of which are dependent on highly skilled user input. In addition, increased automation of induction proofs would be a great gain for the verification systems. Verification tools can be shallower but messier than general mathematical proofs, therefore this area of research is being pursued actively to provide solutions to these problems.

In this thesis, an evaluation methodology was developed and a selected verification system was evaluated for its ability to improve the assurance that systems met certain security objectives. Systems are built to implement some security policy, however the end product can be flawed. For example, security policies can be inconsistent and the implementation of the system may not truly reflect the policies. Formal specifications and verification tools can be used to achieve a more secure system and to provide higher assurance that the system meets the requirements.

Several steps are followed during the assurance process to reveal any unspecified functionality and to create verifiable protection:

(1) Establish a security policy

(2) Develop a Formal Security Policy Model and prove that is meets its own requirements

(3) Create a formal top level specification (FTLS)

(4) Formally map the FTLS to the Security Policy Model

(5) Develop a detailed design of the system

(6) Map the implementation or source code to the FTLS.

First, security policies are created to cover all aspects of protection of organizational assets. Some of these policies are then represented in a Formal[3] Security Policy Model. This model is represented mathematically using precise mathematical logic or a specification language that has formal well-defined semantics. This can be done at several levels of abstraction. A verification system may be used to represent the mathematical model of the security policy.

A formal top level specification (FTLS) is created using notations derived from formal logic to describe assumptions about the world in which a system will operate, requirements that the system will have, and a design to meet those requirements. The

[3] Mathematical

FTLS may also be represented in the language of the verification system. A detailed design of the system is developed from the FTLS using precisely defined semantics.

Proofs are then developed to show that the more concrete levels logically imply the more abstract oriented levels. These proofs are repeatable by third parties. A semi-automatic theorem prover is used to make sure that all of the proofs are valid. Lastly, source code is mapped to the FTLS. This mapping is made possible by the design process. The implementation effectively uses abstraction, layering, and information hiding. The implementation should map to the FTLS so there is no unintended functionality. The five levels used in Formal Methods can be seen in Figure 1.

The FTLS and the security policy model are both represented using the language of a verification system. The verification system examined in this study will be evaluated based on how well it can describe the security policy and system designs, the level of assurance provided, and how easy it is to prove the design implements the policy. At least one simplified version of this complex problem will be established and the verification system will be evaluated based on this sample problem.

B. METHODOLOGY

This thesis was divided into three phases: the desktop analysis phase, the hands-on phase, and the evaluation phase. The initial phase also known as the survey or desktop analysis phase, consisted of developing a list of potential verification system candidates for evaluation and filtering out the verification systems that did not meet a set of broad functional requirements. These requirements were used to construct the set of evaluation criteria with which the " filtering out" process was based. The selected verification systems were the systems that best satisfied the evaluation criteria. The next phase was the "hands-on" study where the empirical study of the eligible verification system was conducted and the third phase was the evaluation of the system and the extrapolation of the results to attempt to apply them to real life systems. In the empirical study, a simplified version of a complex computer security related problem was constructed to illustrate and exercise various problem-solving methods. One security model and one FTLS was formed from this sample problem. Evaluation criteria was established so that during the evaluation process, the results could be measured against the criteria. The evaluation criteria included the previous criteria used in the “filtering out” process as well as additional more specific functional criteria. The selected verification system to be evaluated was run against the sample problem and the results were tabulated based on the evaluation criteria. The purpose of the evaluation criteria was to attempt to reveal any inconsistencies in the chosen verification systems such as the potential ability to prove a false statement was true. The empirical study process can be summarized as follows:

(1) Establish a Security Policy

(2) Develop Mathematical Model of the Security Policy

(3) Represent this Security Policy Model in terms of the verification tool

(4) Develop Formal Top Level Specification (FTLS)

(5) Represent this FTLS in terms of the verification tool

(6) Map the FTLS to the Security Policy Model

(7) Attempt to use the tool to prove that the FTLS implements the policy

In the third phase or the evaluation phase, conclusions were drawn on how well the selected verification systems described the security policy and FTLS. Also, information on how much assurance the selected verification systems provided and how easy it was to prove that the design satisfied the policy was given at the end of the paper. The final process consisted of determining whether the results of the evaluation process were applicable to real world systems.

C. CONCLUSION

Results of the empirical study should determine the usefulness of the verification system for the sample problem. In addition, our work was driven by the need of another project HANNAH (High Assurance Network Authenticator), which involves the creation of a high assurance trustworthy authentication on a network. HANNAH requires the use of a verification system, and therefore relies on the results of the evaluation of the verification systems. This paper attempted to extrapolate from the results a conclusion about the usefulness of a verification system for the verification of real world systems.

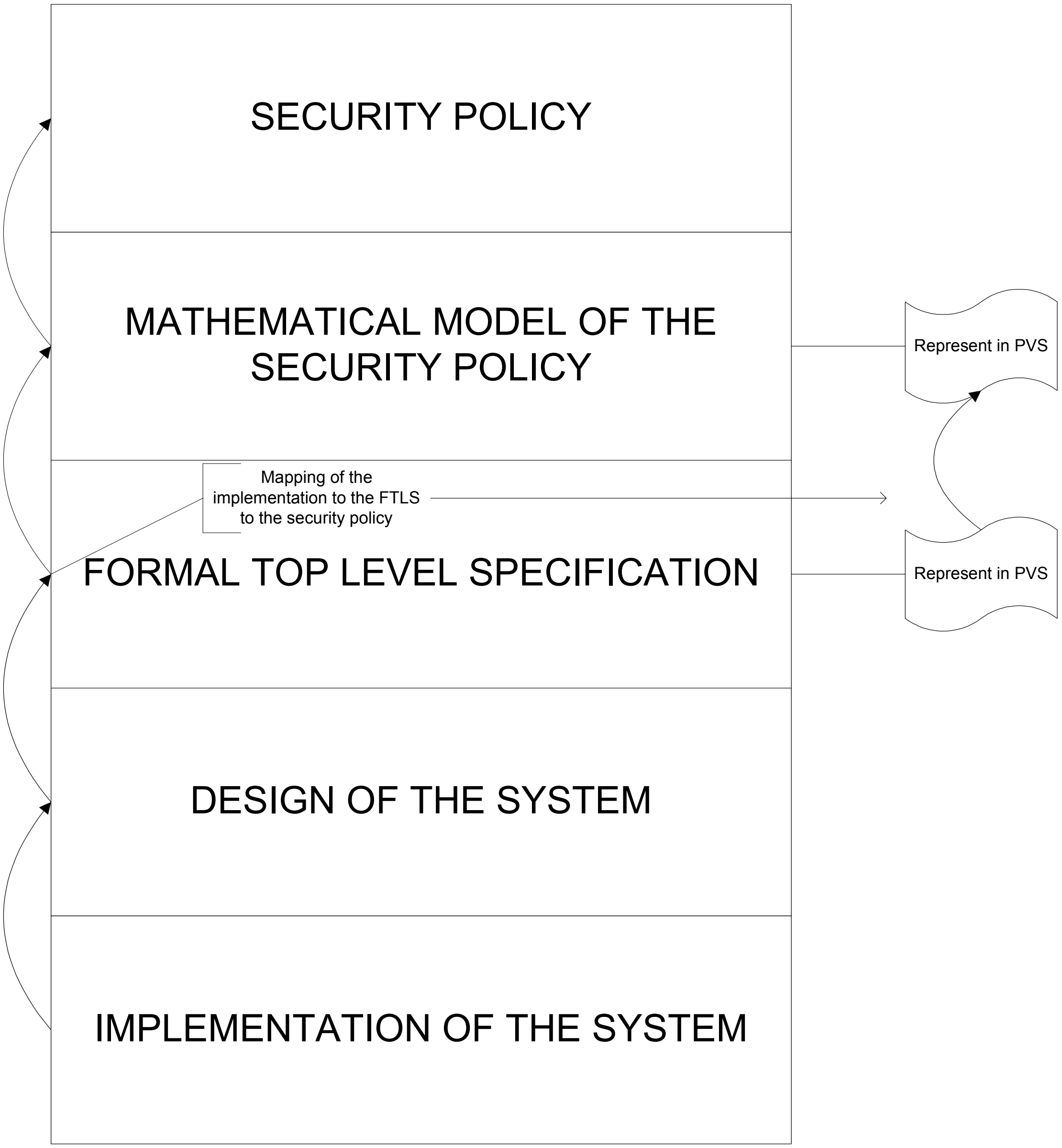

SECURITY POLICY
MATHEMATICAL MODEL OF THE SECURITY POLICY
Mapping of the implementation to the FTLS to the security policy
FORMAL TOP LEVEL SPECIFICATION
DESIGN OF THE SYSTEM
IMPLEMENTATION OF THE SYSTEM
Represent in PVS
Represent in PVS

Figure 1. The Five Levels used in Formal Methods

THIS PAGE INTENTIONALLY LEFT BLANK

II. CANDIDATE ELIMINATION FOR EVALUATION

A. INTRODUCTION

Choosing which verification tools to use and exactly what support is offered from each tool is not always clear. Therefore, there is a need to define criteria so that the tools can be evaluated with respect to their various capabilities. This provides a means of finding which tool is suitable for a specific research environment and what needs of a particular project the tool satisfies. There is no ideal tool for a researcher in formal methods, since the tools come in a vast spectrum of properties and qualifications.

Tools can range from simulators to model-checkers to decision procedures to theorem provers as well as various combinations of these. Theorem provers can be categorized as interactive provers, automated provers, logical frameworks, and inductive provers. Proofs are intended to assert the correctness and provide feedback as well as find errors. In this paper, fifteen verification tools have been preliminary evaluated based on specific evaluation criteria ranging from the "age" of the tool to whether the tool supports multiple levels of abstraction. These tools were chosen as candidates for this evaluation study because of the large amount of information available relating to the tools, the tools' differing qualities, and our specific interests in some of the tools. Here is the list of the evaluated tools. Next to each tool are the tool's homepage and a webpage relating to information about that tool:

- ACL2 (A Computational Logic for Applicative Common Lisp)[Moo03]
  www.cs.utexas.edu/users/moore/publications/km97.ps.Z
- AutoFOCUS [Tum03]
  www.docs.uu.se/ftrtft96/program.txt
- Coq [Inr03]
  www-sop.inria.fr/oasis/personnel/ Simao.Desousa/appsem01.ppt
- Elf/Twelf [Pfe03]
  www.cs.cmu.edu/~rwh/theses/schuermann.pdf
- HOL (Higher Order Logic)[Iyo03]
  http://citeseer.nj.nec.com/hougaard94computability.html
- IMPS (An Interactive Mathematical Proof System)[Far03]

http://imps.mcmaster.ca/doc/imps-overview.pdf

- Isabelle [Pau03]
  http://www.cl.cam.ac.uk/users/lcp/papers/protocols.html
- Nuprl [Cor03]
  citeseer.nj.nec.com/constable86implementing.html
- Otter [Arg03]
  www.cityauditorphilwood.com/warren/sectionf1.html
- PVS (Prototype Verification System)[Sri03]
  http://www-2.cs.cmu.edu/~svc/papers/view-publications-dk02.html
- SpecWare [Kes03]
  http://www.specware.org/documentation/4.0/tutorial/SpecwareTutorial.html
- SteP (Stanford Temporal Prover)[Sta03]
  http://www-step.stanford.edu/papers/railroad.html
- TAME (Timed Automata Modeling Environment)[Arc01]
  chacs.nrl.navy.mil/publications/CHACS/2001/index2001.html
- TPS (Theorem Proving System)[And03]
  gtps.math.cmu.edu/hug93.ps
- Vienna (Vienna Development Method)[Ibm03]
  www.sei.cmu.edu/publications/documents/ 88.reports/88.tr.026.html
- Z/Eves [Ora03]
  www.comp.nus.edu.sg/~dongjs/papers/icfem02dsw.pdf

B. EVALUATION CRITERIA

1. Age

The “age” of a tool can indicate the amount of available documentation of the tool regarding improvements, strengths, weaknesses, purposes, and other information relating to the overall value of the tool. Tools developed for trusted computing efforts may have become obsolete and not been used in a long period of time. A tool can be currently supported and employed, yet be relatively new which leads to concerns involving its acceptability, usability, and proficiency. The more data collected from the research of a verification tool, the more questions that can be answered about that tool. However, even though a tool might be older, it may not be maintained currently, and this could lead to a problem. Therefore, the "ideal" tool should be mature as well as be currently maintained and supported. From the list of fifteen tools, the ones that have only been developed within the last seven years are AutoFOCUS, Coq, Elf/Twelf, and TAME. However, out of the complete list of tools, all are being used in some kind of research environment currently at either a university (e.g Stanford), laboratory (e.g Carnegie Mellon University), or a company (e.g ORA Canada). ACL2, Coq, Z/Eves, Isabelle, PVS, and Nuprl all have current versions released in 2002. [Appendix A]The developers who are currently supporting each tool can be found in Appendix L.

2. Purpose

A tool's purpose plays a major role in determining how effective it will be when used for a particular project. The key qualifications of the tool should be its ability to ensure that the system satisfies some set of security properties as well as being able to express software properties that are of our interest. If a tool is used exclusively for the purpose of satisfying a particular mathematical concern, then its usefulness is questionable. ACL2, PVS, Z/Eves are all general purpose theorem proving tools that can be employed in almost any environment from small software projects to industrial-size verification projects. HOL has a wide variety of uses from formalizing pure mathematics to verification of industrial software. The Vienna Development Method (VDM) supports the top-down development of software systems and STeP is a tool for the computer aided formal verification of reactive systems. ACL2, PVS, Nuprl, Coq, and Isabelle all have the ability to tackle real-world problems primarily due to the improved capabilities of interactive systems. These systems are used for purposes other than verification. IMPS provides organizational and computational support for the traditional techniques of mathematical reasoning. TPS has facilities for searching for expansion proofs and translating them into natural deduction proofs as well as constructing and translating natural deduction proofs. However, its usage in real-world problems and describing properties of software has not been documented. Otter's main application is in abstract algebra and formal logic and has been used to answer questions in the areas of finite semigroups. However, it also deals more with mathematics and can be difficult to use to illustrate the properties of software. [Appendix B]

3. Implementation Language

A tool whose implementation language can be run on many systems becomes a more usable and effective tool, due to its portable nature. Therefore, if the tool's implementation language falls in the category of the more "popular" languages (e.g. ML, Common Lisp, Java, or C), then the likelihood of it being able to run on a user's system is higher. This reduces the amount of effort that includes time, money, and research to find an appropriate environment for the tool. All fifteen tools satisfy this quality, since each tool's implementation language is a "popular" language. [Appendix C]

4. Resource Requirements

A tool should be able to run on a specific platform that is currently available at a user's work environment. Otherwise, the cost and time to implement a suitable system architecture for the tool increases significantly. Therefore, the tool should be able to run on the more familiar operating systems (e.g. Windows and Unix). It is also advantageous if the tool can be run on different platforms. ACL2, Otter, Isabelle, IMPS, Coq, TPS, AutoFOCUS, Vienna, and HOL can all be run on Unix. TPS, PVS, Z/Eves, Coq, ACL2, and SteP can run on Linux. Coq, Z/Eves, and HOL also run on Windows machines and SteP, PVS, and Z/Eves also run on Solaris. Nuprl can run in any Common Lisp with CLX. Therefore, all the tools run on common platforms, and some can even be run on more that two different operating systems. [Appendix D]

5. User Friendly

The amount of effort and control that is required by the user to achieve a result can determine if a tool is "user-friendly". The degree of automation can affect the level of difficulty of using a verification tool. Fully automatic tools may only need a "push of button" and the user just has to wait for the answer. However, these tools are limited in what they can verify. In addition, users still have to set prover parameters and generate prover commands to pilot the theorem prover. Otter is fully automatic and has more than a hundred Boolean and numeric parameters. For the prover to perform well, these parameters have to be set up correctly, which increases the user's time and energy spent on operating the tool. Interactive theorem provers (a.k.a. "proof checkers") are driven by the human user, and therefore completing the proof is slower and requires more effort by the user. However, interactive provers can achieve parts of a proof by itself, and the user can receive assistance from people with expertise in using interactive theorem provers. The key qualification that a "user friendly" tool possesses is the ability to use it without having to know the language or mechanics of the tool in depth or training to become fluent. The beginner should only need to know a basic knowledge of the tool and a small repertoire of commands. TPS and SteP can be both proved automatically or interactively, however TPS is quite primitive in automatic mode. ACL2 is a semi-automatic inductive prover where the user's responsibility is to understand the mathematical logic, be able to construct a proof interactively, and have the mathematical insight of why the model has a desired property [Define inductive prover]. Coq, Z/Eves, Nuprl, PVS, IMPS, and HOL are all interactive theorem provers. Z/Eves can automatically perform large proof steps, yet those steps can by finely directed by the user. With Nuprl, it is impossible to develop an incorrect proof and users can write proof generating programs. PVS is much quicker than HOL, and HOL users have to first learn some ML before they can effectively use the tool. TAME, an interface to PVS, provides a way for users to create proofs using "natural" or automatic proof steps, without having to learn the details of the PVS proof steps. Also, TAME provides better user feedback in comparison to PVS. Lastly, the descriptive language of Vienna is relatively difficult to learn when trying to employ the tool, and Elf/Twelf is not suited for the interactive development of theories. When deciding which tool to use based on whether the tool is "user friendly", the ultimate choice is dependent on the task at hand. The user may want fine control of the prover to investigate proof strategies or proof failings. On the other hand, the task may require beginners to use the tool right away, and therefore a tool that requires the least amount of knowledge to use it is needed. Therefore, the significance of how "user friendly" a tool basically depends on the needs of the user. [Appendix E]

6. User Interface

As a supplement to Section 5, the user interface is a more specific attribute that helps to categorize the tool by ease of use. A "point and click" graphical user interface is much simpler to use than an interface that requires the user to remember 1000 different commands. Z/Eves and AutoFOCUS use a graphical interface and TAME also provides a friendly user interface. Nuprl, SteP, and Otter use X-Windows, ACL2, Elf/Twelf, PVS, IMPS, and HOL use Emacs, and Isabelle uses Ml as their user interface. Coq, however, does not use any special interface such as Emacs and has a standard teletype-like shell window. [Appendix F]

7. User Presentation Language

This section is also a supplement to Section 5. The user presentation language is an important attribute of a verification tool and is entirely dependent on the user's preferences, background, and needs. If a user is not familiar with the tool's presentation language, then problems can emerge. The user may have to spend an indefinite amount of time to become knowledgeable about the language and the likelihood of mistakes maybe higher due to inexperience. It is best if the tool's interface language is one that the user is comfortable with and competent at using. We have not explored the information regarding this specific criteria yet, therefore the various tools' presentation languages will be presented in Phase 2. [Appendix G]

8. Consistency of Specifications

Verification tools may be used to find errors and inconsistencies and can determine the level of confidence of the correctness of a system. However, the tools themselves should first be shown to be correct, otherwise what the tool outputs may not be trusted. Any inconsistencies in the underlying logic of the verification systems such as the potential ability to prove a false statement is true can be detrimental. Logic that is more familiar to users leaves less room for unexpected inconsistency problems. A specification is a list of properties that a user expects from a program. When a specification includes axioms, the tool should allow new axioms to be added while maintaining consistency with the original set. Managing inconsistent specifications includes activities such as consistency checking, reasoning, and analysis. A consistency checker can expose missing cases, unwanted non-determinism, and other application-independent errors. Checking a specification for unwanted nondeterminism and missing cases can be computationally expensive and complex otherwise. ACL2, Z/Eves, Nuprl, PVS, TAME, Isabelle, and HOL all have some form of consistency checking. Whether Coq and Elf/Twelf support consistency of specifications is unclear. More research will be conducted due to inadequate information regarding consistency of specifications for the remaining verification tools. However due to limited documentation, conclusions may still not be apparent in Phase 2. [Appendix H]

9. Executable Specifications

When a specification is executable, a 'feel' for the system can be gained. Executable specification allows one to demonstrate the behavior of a software system before it is actually implemented. Therefore, specifications are constructive, since they not only demand the existence of a solution, they actually construct it. Through executable specifications, a user can modify and test specifications to achieve some expected result. Each specific property can produce several examples that uphold this property, and the user can then instantiate theorems via the generalization of these examples. ACL2, Coq, Nuprl, Vienna, and HOL all support executable specifications. Information on whether the remaining verification tools also possess this attribute will be presented in Phase 2. [Appendix I]

10. Multiple Levels of Abstraction

Formal Specification and Verification Tools can be used to achieve a more secure system and to provide higher assurance that the system meets the requirements. Several steps are followed during the assurance process to reveal any unspecified functionality and to create verifiable protection. The initial steps involve developing a security policy and a top level specification of the system. Security policies are created to cover all aspects of protection of organizational assets. Some of these policies are then represented in a Security Policy Model. This model is represented mathematically using precise mathematical logic or a specification language that has formal well-defined semantics to specify the system. A verification tool may be used to represent the mathematical model of the security policy. A formal top level specification (FTLS) is created using notations derived from formal logic to describe assumptions about the world in which a system will operate, requirements that the system will have, and a design to meet those requirements. The FTLS is also represented by a verification tool. The FTLS and the Security Policy Model are at different levels of abstraction. Therefore, the verification tool should support multiple levels of abstraction so that one can prove that the top level specification of the system satisfies the security model. ACL2, PVS, and HOL support multiple levels of abstraction. Phase 2 will determine whether the other verification tools satisfy this property or will remain questionable due to limited documentation. [Appendix J]

11. Expressiveness

The degree of expressiveness of a verification tool can determine its usefulness when applied to more complex and difficult problems. A tool's expressiveness can be dependent on the logic it uses as well as its unique additional features. Logic, a formalism for representing specifications, can be propositional, first order predicate, higher order, temporal, etc. Higher order logic can express the more complicated properties in comparison to first order logic. However, one should primarily address whether the tool has the ability to sufficiently express properties of software systems, rather then its underlying logic. ACL2, Z/Eves, and Otter can only be expressed with first order logic. Coq, Nuprl, PVS, Isabelle, IMPS, TPS, TAME, and HOL are based on higher order logic. Since ACL2 is programmed in the same logic it supports, this ensures that the logic is a practical means of building large formal systems. Coq is an extension of the Calculus of Constructions with inductive types and is well adapted to inductive reasoning. Elf/Twelf is a logical framework (LF) based on predicative type theory. Z/Eves has unique attributes that include an expressive formal specification and programming language, practical automated deduction support, and mathematical soundness. One feature of Nuprl is that the logic and system take account of the computational meaning of assertions and proofs. PVS can introduce axioms freely and Isabelle can support reasoning in several object logics. AutoFOCUS is based on simple temporal logic and the logic underlying the semantics of Vienna is based on the Logic of Partial Functions. Due to the various different attributes of each tool, once again there is no ideal tool, and decisions should be based on the needs of the user and the task at hand. [Appendix K]

Evaluation Criteria	***Summary Statement***
Age	*A tool should be old enough and currently maintained*
Purpose	*A good tool should be versatile to verify a variety of mathematical problems and be able to express software properties*
Implementation Language	*The more systems a tool's implementation language can run on, the better it is*
Resource Requirements	*The larger number of common operating systems a tool can run on, the better it is*
User-friendly	*A good tool should take the guess-work out and make its operation simpler and more flexible to the user*
User Interface	*A GUI tool is better than a non-GUI tool*
User Presentation Language	*A user should be familiar with a tool's presentation language*
Consistency of Specifications	*A good tool should have consistency checking for an entire set of tailorable specifications*
Executable specifications	*A good tool should have the capability of executable specifications, so the user can get a "feel" of the system*
Multiple Levels of Abstraction	*A tool should be able to support multiple levels of abstraction to provide verification that the top level specification satisfies the security policy*
Semantics	*A tool's underlying logic and other unique functions can play a role in how expressive it is, however the main concern should be whether the tool sufficiently expresses the properties of software systems*

Table 1. Synopsis of Evaluation Criteria

C. PHASE 2: SELECTED CANDIDATE: PVS

The amount of available documentation, the number of satisfied evaluation criteria, and familiarity with the tool (See Appendices A - L), resulted in PVS being selected as the verification system to be evaluated in the empirical study. Due to time constraints, some verification systems could not be fully evaluated in the survey by specific evaluation criteria such as multiple levels of abstraction, consistency of specifications, executable specifications, and user presentation language. The proper methodology for analyzing those tools in the empirical study is as follows:

- Express the mathematical model of the security policy by the verification system
- Express the formal top level specification by the verification system
- Illustrate interlevel mapping between the formal top level specification and the mathematical model of the security policy

A large amount of time is needed to learn a verification tool's specification language, its underlying logic, and especially its theorem proving capabilities. To be able to evaluate a verification system to see if it supports multiple levels of abstraction, consistency of specifications, and executable specifications as well as determining its user presentation language requires an ample amount of time. Installation of the verification tool, understanding and writing the various specifications, and proving the required theorems are just some of the activities that need to be considered before proper evaluation can be done.

D. CONCLUSION

ACL2 was also considered an eligible candidate for study since it satisfied most of the evaluation criteria and also came with sufficient documentation. Due to time limitations, the formal security policy model as well as the formal top level specification (FTLS) could not be expressed in ACL2. However, a background of ACL2 containing information about its theorem prover, underlying logic, and special features as well as a brief comparison between PVS and ACL2 is provided in this paper. In future research, ACL2 will be evaluated following the same methodology introduced in this paper to evaluate PVS. A more in depth view of the process of our empirical study of a verification tool can be found in Chapter VI, Section E.

III. FORMAL SECURITY POLICY MODEL

A. INTRODUCTION

Formal methods is a rigorous method of analyzing the trustworthiness of a system. This technique is based on the use of models. With mathematical rigor, it can establish certain properties about a system. The goal of formal methods in the development of secure systems is to show that a security policy is consistent and completely enforced by the modeled system. The goal of a security policy is to "protect the confidentiality, integrity, and availability of the system against attacks by malicious users and mistakes made by innocent users"[Sec98]. In order to provide an adequate level of protection, a computer system has to enforce an appropriate set of policies and to enforce them correctly. The security model is a precise and unambiguous statement of a system's security policy. It is an obvious representation of the policy. Some important security models are The Bell and LaPadula model [Bel73], a confidentiality model and the Biba model [Bib77], an integrity model, both of which are lattice models, characterized by systems whose labels can be arranged in a lattice dominance hierarchy.

The formal security policy model is a high level model of the Reference Monitor abstract reference and authorization functions. It is a mathematically precise statement of a security policy. The formal model enumerates operations that can be invoked to observe or modify that state. Also, it must be specific about critical technical elements such as what preconditions must be true for the operation to begin, what post-conditions will be true when the operation ends, and exactly what changes of the identified state will occur. The formal model needs to be abstract where the system state is divided into explicit and implicit parts. The specification describes pre- and post condition, and effects, only on explicit state. Therefore, non-determinism is permitted (satisfying pre-conditions is not a guarantee of execution). The formal model is kept as simple as possible, and detail is added only when it is required to prove that the implementation satisfies the initially stated requirements and security properties. As a consequence, the specification may contain non-deterministic transitions; one purpose of refinement is to resolve non-determinism in a way that is consistent with the initial properties and requirements [Sec96]. The formal model is of a system interface, in other words it is an abstract model of something real and it is written in a formal specification language. [Irv03]

The formal security policy model consists of two parts:

- A general model of a system (system as a state plus a set of operations)
- A definition of a secure system – Each formidable clause in the security policy is turned into a predicate that constrains the system in some way. The overall definition of a secure system is one where all the constraints are satisfied.

The model must represent the initial state of a system, the way in which the system progresses from one state to another, and a definition of a “secure” state of the system. To be acceptable as a basis for a TCB, the model must be supported by a formal proof that if the initial state of the system satisfies the definition of a “secure” state and if all assumptions required by the model hold, then all future states of the system shall be secure.

B. NON-DISCRETIONARY ACCESS CONTROL

Non-Discretionary Access Controls are access controls based on rules and information associated with the subject and object where the information is called a security level and the levels are implemented as Labels. Label-Based policies in which objects (files, directories, etc.) and users of a computer system have security labels can confine malicious activities such as Trojan Horse Attacks. When a system supports labeled data and labeled users it can reflect a real picture of what sensitive and non-sensitive information there is on a system. Files are labeled with a fixed access class value and users can execute at an access class level that is constrained by a user’s clearance level. This label comparison is the basis of the access control. Mandatory Access Control (MAC) is the most commonly used non-discretionary access control. Under MAC, if a subject label and the object label cannot be compared then access is denied, however if they are comparable, access is determined based on rules regarding the relationship between the labels. [Sul03]

C. THE BELL AND LAPADULA MODEL

The Bell and LaPadula model (BLP) [Bell73] has been the most influential model of security over the past ~30 years. The policy in the BLP model and some of the elements of the model are embedded within the Trusted Computing System Evaluation Criteria (TCSEC) also know as the "Orange Book"[Sul03]. The Bell and LaPadula model, a confidentiality model, has three main axioms:

- BLP Axiom 1

  Simple-security property(SS): a subject *s* is allowed to read an object *o only if* the security label of *s* dominates the security label of *o*

  - No read up
  - Applies to *all subjects*

- BLP Axiom 2

  **-property*: a subject *s* is allowed to write an object *o only if* the security label of *o* dominates the security label of *s*

  - No write down

- BLP Axiom 3

  Discretionary Security Property (ds): A state satifies the "ds" property if for each member of the current access set, the specified access mode is included in the access matrix entry for the corresponding subject-object pair [Sul03]

  - Allows an individual to extend access to an object to anyone that is allowed to observe the document under the SS and the '*' property
  - Can only reduce the set of reachable states

The security policy that will be used in this paper will be very similar to the Bell and LaPadula model. Each subject will be given a fixed security label and each object will also be given a fixed security label.

D. CONSTRUCTING OUR FORMAL SECURITY POLICY MODEL

1. Security Policy

- Property 1 (same as BLP Axiom 1)

Simple-security property(SS): a subject *s* is allowed to read an object *o only if* the security label of *s* dominates the security label of *o*

  - No read up
  - Applies to *all subjects*

- Property 2

  Property 2: a subject *s* is allowed to write an object *o only if* the security label of *o* is equal to the security label of *s* (restricted special case of BLP Axiom 2)

  - No write down

- Property 3

  Property 3 (Tranquility Principle): the static security level of a subject may not change

Tranquility restricts the changing of security labels of subjects and objects. Strong tranquility is where security labels of subjects and objects never change during an operation. Weak tranquility is where security labels of subjects and objects never change in such a way as to violate the security policy. Our model assumes strong tranquility. The advantage is that the system state always satisfies security requirements, however the disadvantage is that the system becomes less flexible.

2. The Basic Security Theorem

When writing a formal model or specification, “the general validation approach is to state and prove theorems about the model that one intuitively expects to be true”[Irv03]. Inconsistencies, unintended functionality, and mistakes in the specification are uncovered when proving the test theorem is true. This theorem is appropriately named the Basic Security Theorem or the Fundamental Theorem and takes the form of a safety property or invariant. The real value provided by the Basic Security Theorem

(BST) is that the security can be demonstrated to be an inductive property where a change in state of one object can affect the state of other objects which in turn effects the whole system. "The specification process utilized in the BST can prove extremely valuable when attempting to map the objects of an information system and the vulnerabilities that are associated with these objects" [Kav02]. The BST provides the following properties:

A system is secure iff

- Its initial state is secure
- Each action that starts in a secure state results in a secure state

In our model, a secure state satisfies properties one through three.

3. Anatomy of our model

a. Elements

- Subjects: active entities (users, processes,…)
- Objects: passive entities (data, files, directories,…)
- Access Attributes {read, write}
- Security Levels (Top Secret, Secret, Classified, Unclassified)

b. Components

- Current Access set
- Object hierarchy
- Access matrix
- Security Level function – Mapping Subjects and Objects to Security Labels

c. Properties

- The system should satisfy Property 1, 2, and 3 from above

d. Rules

- State transition operators

e. Theorems and proofs

- Justifications and proof
  - o Basic Security Theorem

E. THE MATHEMATICAL MODEL OF THE SECURITY POLICY

- Each subject and object is assigned a security label

Let slSubject be a function from subjects to security labels (*slSubject: Subject -> SL*) and slObject be a function from objects to security labels (*slObject: Object -> SL*). In addition, let SL be a set of security labels. The security labels are classified as {Top-Secret, Secret, Classified, Unclassified}. Assume that SL is totally ordered and finite.

Subject	***SL***
s_1	*TS*
s_2	*S*
.	
.	
.	
s_i	*C*

Table 2. Subjects to Security Labels

Object	***SL***
o_1	*S*
o_2	*TS*
.	
.	
.	
o_i	*C*

Table 3. Objects to Security Labels

- Let there be *i* subjects and a specific subject is represented by s_i
- Let there be *j* objects and a specific object is represented by o_j
- Let M be a set of distinct modes of access (rd, wr) between a subject and an object
- Let the current access authorizations (CAA) be a function from

  Subject X Object -> M. CAA represents the current access set of the system.
- Example of the authorization matrix

  This authorization matrix represents the current access authorizations (CAA) of subjects to objects and therefore the constraints on information flow. The read privilege is represented by r and the write privilege is represented by w.

Object

		o_1	o_2	.	.	.	o_j
	s_1	*rw*	*r*		*rw*		
Subject	s_2		*r*	*rw*			*r*
	.			*r*		*rw*	
	.		*rw*				*rw*
	.	*rw*			*r*	*r*	
	s_i	*rw*		*rw*		*r*	

Table 4. Subjects to Objects

- The Mathematical Model of the Security Policy
  - o CAA(s_i, o_j, r) $\rightarrow$ slSubject(s_i) $\geq$ slObject(o_j)
  - o CAA(s_i, o_j, w) $\rightarrow$ slSubject(s_i) = slObject(o_j)

F. CONCLUSION

A security policy is a statement of the security we expect the system to enforce. The security policy model can represent a security policy or a set of policies. It focuses on characteristics of policies by abstracting away detail. Generally, the security policy model is stated in a formal language such as mathematics. Mathematical logic separates and clarifies important components of the system. Given the complexity of modern computer systems, it is no wonder that models of some form or another are used in an effort to gain an intimate understanding of their inner workings [Kavanagh].

IV. FORMAL TOP LEVEL SPECIFICATION

A. INTRODUCTION

A formal top level specification (FTLS) is a top level specification that is written in a mathematical language to allow theorems showing the correspondence of the system specification to its formal requirements to be hypothesized and formally proven. It is also a description of the system that shows those system changes that may interact with the security properties of the system. The components of a classical FTLS include a mechanism for describing the security portions of the system state and the state transitions of the system that are security relevant. The goal of the FTLS is to prove that if the system starts in a secure state, then it will never enter an insecure state using the allowable state transitions.

The FTLS shows the actions of the system including all the exceptions and should characterize inputs, outputs, and effects. There should be a coherent mapping between inputs and outputs. Processing takes a set of inputs applies them to the current state and results in outputs and effects. The effects are the changes to the internal state. It may be the "side effects" of the state changes or error handling. Therefore, the elements of a state include the effects that were created from processing. Processing also reports errors that occur in the system that are not shown in the effects. Error checking and reporting is important since in the normal case, state changes should not be made if there are any errors in the system. In addition, these errors such as a subject accessing an object that does not exist can lead to a potential covert channel. The FTLS is an abstraction of all the processing and the effects. "Writing an FTLS is valuable because many behaviors of the system that have an impact on security can easily be overlooked in a less formal description. This is particularly true of behaviors that might be considered side effects of operations that have some other primary purpose" [Sec96*].

When constructing a FTLS, an interface specification for the implementation should first be developed. The implementation consists of a number of security-related and non-security related entry points. A mapping should then be constructed from the interface to the FTLS class. Each security related entry point in the implementation should be mapped to a transform function in the FTLS which is turn should be mapped to a transform function in the model. Therefore, the interface can be characterized in terms of transforms. The non -security related entry points should be mapped to some "no_operation" transform in the FTLS that should be mapped to a "no_operation" transform in the model.

A system cannot make changes without inputs, therefore there are inputs associated with each entry point. Specifying all the entry point accounts for all the activities and state changes that can occur in that system. Therefore, the FTLS can completely characterize the implementation system. Since mapping the implementation to the formal security policy model can be rather difficult, the FTLS provides an abstraction of the properties of the implementation that in turn can be mapped to the properties of the formal security policy model(See Figure 2).

In conclusion, the formal top level specification of the TCB (Trusted Computing Base) should accurately describe the TCB in terms of exceptions, error messages, and effects. The FTLS should be an accurate description of the TCB interface and describe the operations that the TCB provides at its interface and the information that those operations return which are dependent on the internal state of the TCB. In other words, the FTLS should specify accurately all operations and the effects of those operations on all system structures that it might have modified and that are visible to the user. In addition it should specify accurately the effects of those operations on all structures associated with other subjects that can affect the error messages and return values that those subjects will subsequently receive. The FTLS must support three main goals. First, it must support a proof that the system design enforces the security policy. Secondly, it must provide a basis for a catalogue of all covert storage channels. Lastly, it must provide a criterion of correctness for the implementation [Rad87].

B. CONSTRUCTION OF THE FORMAL TOP LEVEL SPECIFICATION(FTLS)

The FTLS is a complete description of the behavior of the system at a particular level of abstraction. It is complemented by a user interface specification that gives a more concrete description of the appearance of the system. Many implementation details of the computer system are not represented in the FTLS. The process of developing the FTLS is as follows:

1. Represent the State Elements of the Implementation as State Elements of the FTLS

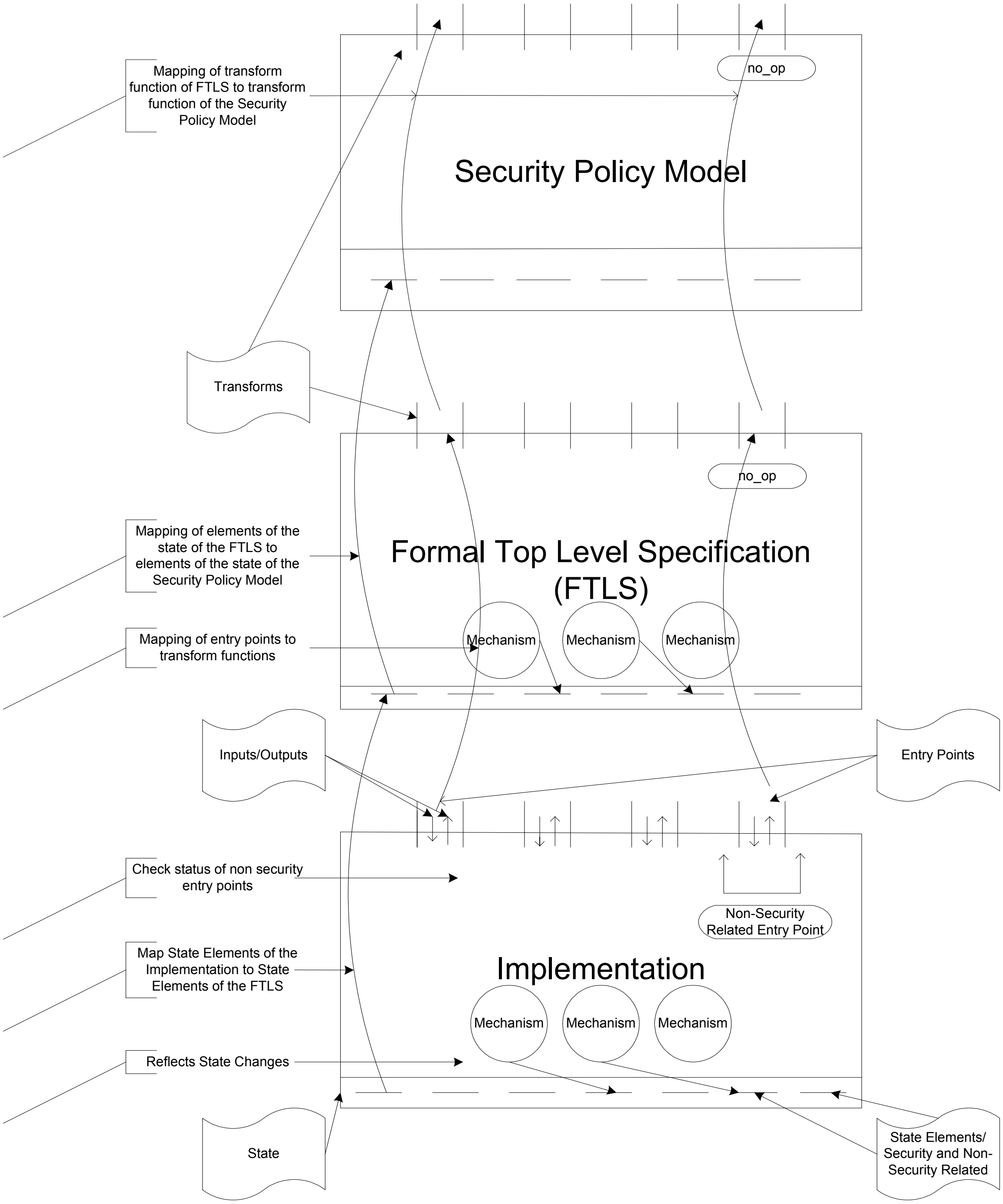
The security-significant state elements of the implementation of the computer system may be viewed abstractly as processes, and memory blocks, and their relationships. In a system, inputs are called requests and outputs are called decisions. The system consists of all sequences (request, decision, state) with some initial state. Each process and memory block will have a fixed associated security label. In addition, a process will be associated with a set of memory blocks, each with a given access mode (read or write) for that process. Each process will be identified through an identification number appropriately labeled as ProcessID. Two processes can share the same set of memory blocks and a process can be associated with a different set of memory blocks and different access modes for those memory blocks at various times(See Figure 3).

2. Represent the Entry Points of the Implementation as Transform Functions of the FTLS

With respect to this FTLS and Model, the security-significant entry points of the computer system are those for the opening and closing of data objects (e.g., files).The corresponding FTLS transform functions are adding a memory segment with a specific mode to a process and deleting a memory segment with a specific mode from a process(See Figure 3).

C. CONCLUSION

The security policy model and the formal top level specification are simple models whose primary purpose is to illustrate the inter-level mapping problem. Therefore, the level of detail in both these specifications will be expanded in future research. For example, information flow is not modeled in our specifications since we assume that it is handled elsewhere. This is the difference between this model and a non-interference model. In our specifications, we model “open/close” of a subject accessing an object, however we do not model “read/write”. Clearly, there are many aspects of our policy that could be implemented in our model, however due to the limited amount of time, our main focus was on demonstrating the inter-level mapping problem.



Mapping of transform function of FTLS to transform function of the Security Policy Model
no_op
Security Policy Model
Transforms
no_op
Mapping of elements of the state of the FTLS to elements of the state of the Security Policy Model
Formal Top Level Specification (FTLS)
Mapping of entry points to transform functions
Mechanism
Mechanism
Mechanism
Inputs/Outputs
Entry Points
Check status of non security entry points
Non-Security Related Entry Point
Map State Elements of the Implementation to State Elements of the FTLS
Implementation
Mechanism
Mechanism
Mechanism
Reflects State Changes
State
State Elements/ Security and Non-Security Related

Figure 2. Detailed Mapping of the Multiple Levels

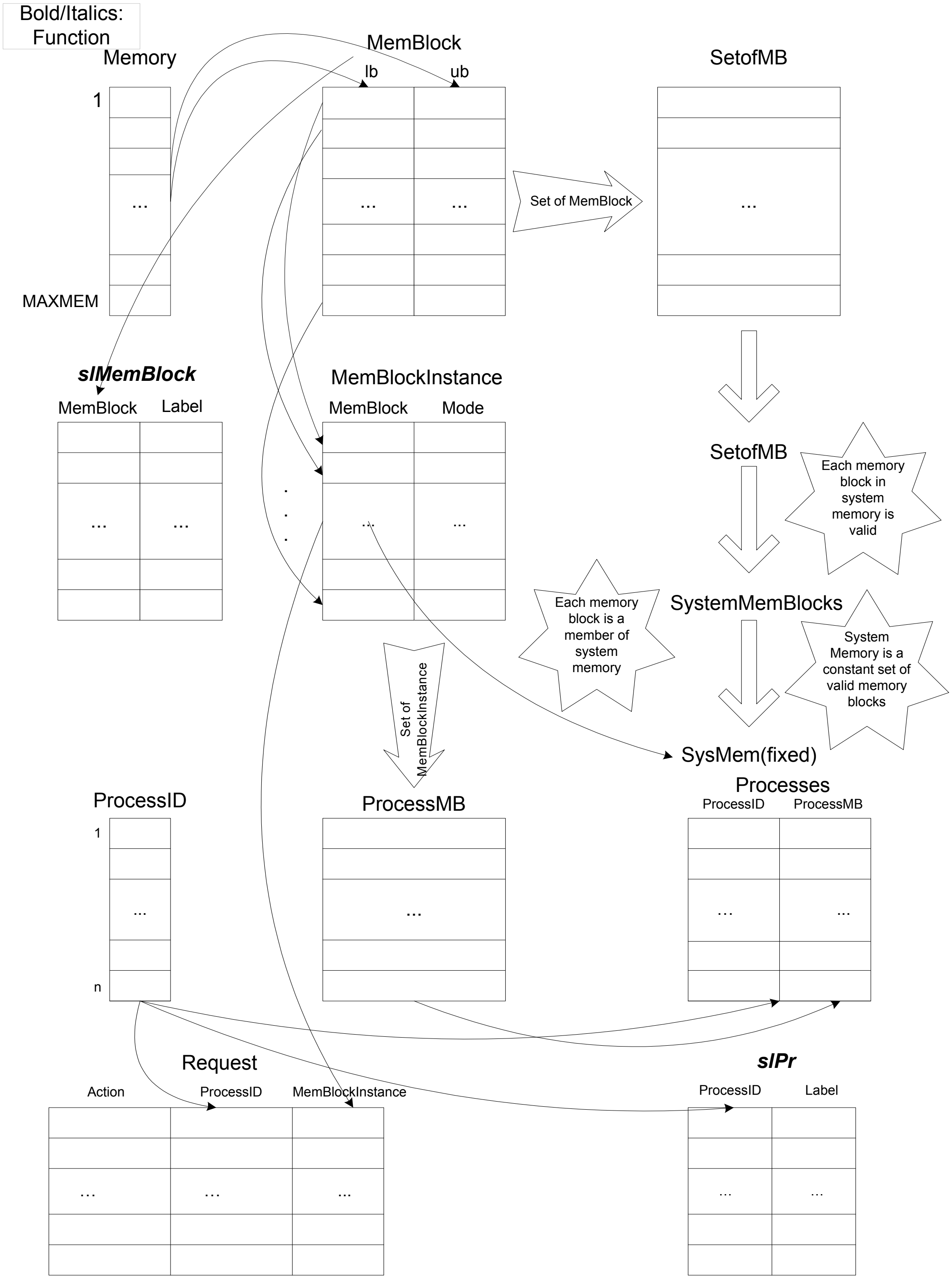


Figure 3. Representation of FTLS

THIS PAGE INTENTIONALLY LEFT BLANK

V. THE PVS SPECIFICATIONS

A. INTROUCTION TO PVS

The Prototype Verification System (PVS) can be defined as a verification system that provides an interactive specification/verification environment for writing formal specifications and verifying formal proofs. This system lets the user interactively guide the proof construction and provides powerful theorem-proving capabilities as well as an expressive specification language. One of its most special features is the use of various decision procedures for different logical domains.

1. Semantics

The semantics of PVS consists of classical higher-order logic and includes predicate subtypes, dependent typing, and parameterized theories. The basic logic in PVS is propositional logic extended with equational logic. Abstract data types such as lists and trees can be defined and axioms can be introduced freely. Definitions, which allow for conservative extension can be recursive and can include inductively defined predicates and recursively -defined abstract data types. Standard PVS types include:

- numbers
- tuples
- arrays
- records
- functions
- sets
- sequences
- lists
- trees

2. Typechecking

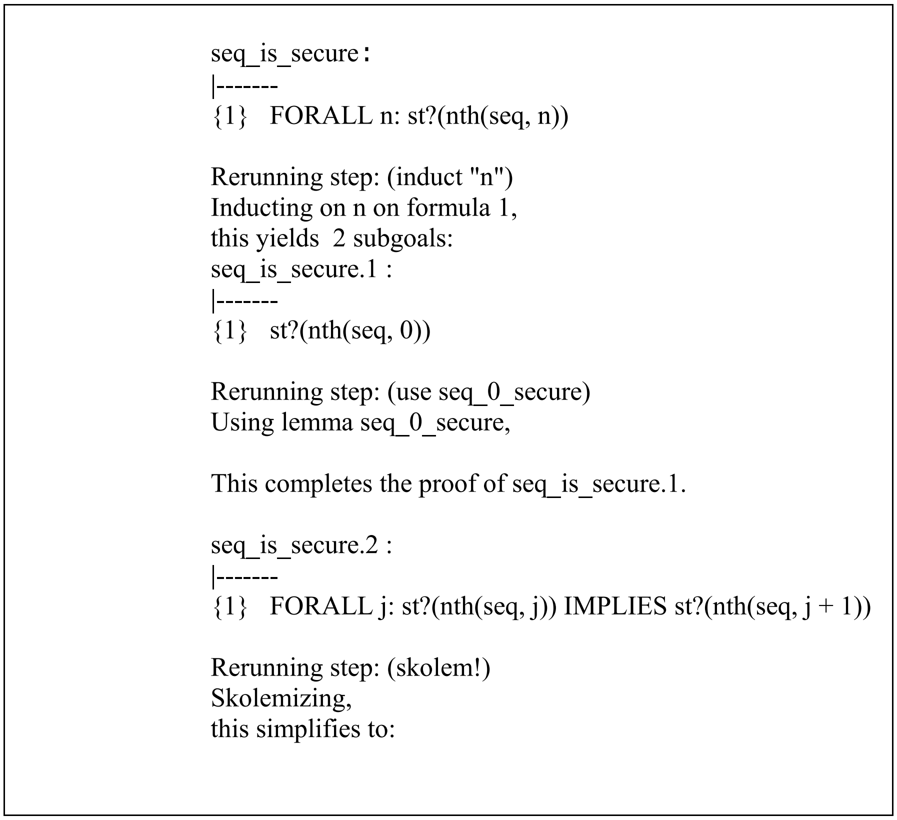
Typechecking is another feature offered by PVS that resolves name references, introduces user-specified type conversions, and makes sure that the types are consistent within the specification. The typechecker will generate proof obligations that may have to be proved using the interactive prover, however most can be dismissed automatically. These proof obligations allow refined typing to be asserted through subtyping judgments. This provides strong checks on consistency and other properties. The type system of PVS is undecidable, so typechecking is not completely automated.

3. Theorem Prover

PVS' interactive theorem prover/proof checker consists of numerous proof commands that can be placed into individual categories[1]:

- Annotation
- Control
- Structural
- Propositional
- Quantifier
- Equality
- Using Definitions and lemmas
- Extensionality
- Induction
- Simplification using decision procedures and rewriting
- Installation and Removal of rewrite rules
- Making type constraints explicit
- Model Checking
- Converting a strategy to a rule

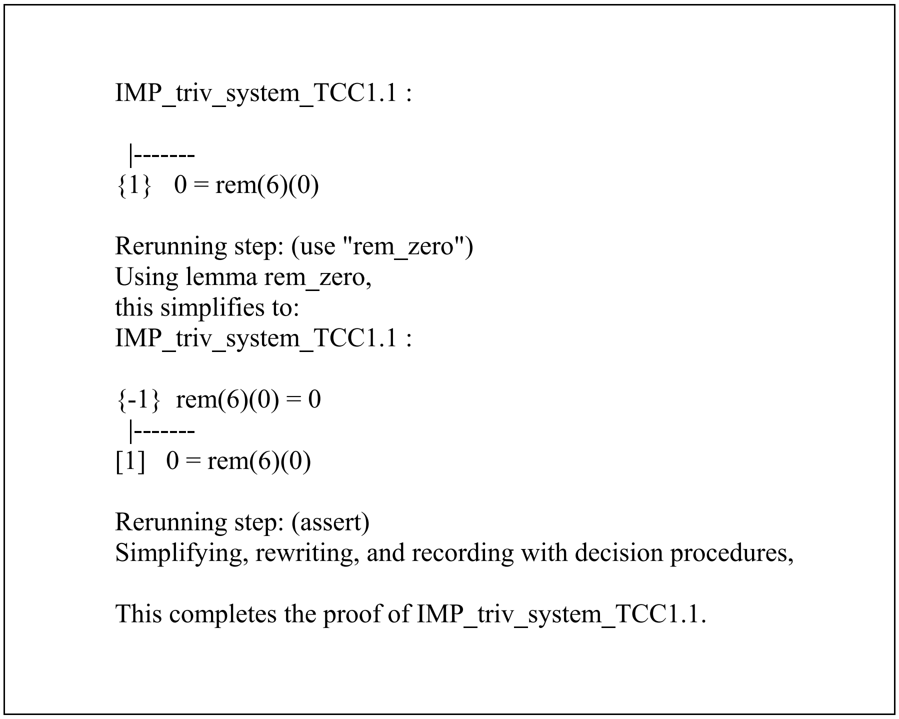
Execution of a certain command can create subgoals or complete a goal. Therefore, the proof checker becomes the manager of the construction of the proof by prompting the user to give adequate commands for a certain subgoal. For example, in Figure 4, the goal is to prove *sec_is_secure* which is to prove that for a sequence of states, every state in the sequence is secure. By induction, the goal to is split into two distinct subgoals where the first is proving the initial state is secure. The second subgoal is to prove that if *n* is not the initial state and by hypothesis the prior state was secure, then all reachable states after the nth state are secure.



```
seq_is_secure:
|-------
{1}   FORALL n: st?(nth(seq, n))

Rerunning step: (induct "n")
Inducting on n on formula 1,
this yields  2 subgoals:
seq_is_secure.1 :
|-------
{1}   st?(nth(seq, 0))

Rerunning step: (use seq_0_secure)
Using lemma seq_0_secure,

This completes the proof of seq_is_secure.1.

seq_is_secure.2 :
|-------
{1}   FORALL j: st?(nth(seq, j)) IMPLIES st?(nth(seq, j + 1))

Rerunning step: (skolem!)
Skolemizing,
this simplifies to:
```

Figure 4. Proofs of Goals and Subgoals in PVS

PVS comes with a number of predefined theories (i.e.; proven "goals") and its theorem prover automates most of the low-level proof steps. The prover also consists of a powerful collection of inference steps that include Boolean simplification, arithmetic and equality decision procedures, and automatic rewriting.



```
IMP_triv_system_TCC1.1 :

  |-------
{1}   0 = rem(6)(0)

Rerunning step: (use "rem_zero")
Using lemma rem_zero,
this simplifies to:
IMP_triv_system_TCC1.1 :

{-1}  rem(6)(0) = 0
  |-------
[1]   0 = rem(6)(0)

Rerunning step: (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of IMP_triv_system_TCC1.1.
```

Figure 5. Using a Pre-defined Lemma from the PVS Prelude File

For example, in Figure 5, the goal is to prove that 0mod6 = 0 which is written *0 = rem(6)(0)* in PVS. The *use* command invokes lemmas with instantiation. Therefore, in this example, a lemma called "rem_zero" defined as *rem_zero: LEMMA rem(b)(0) = 0* for all positive integers b and found in the *modulo_arithmetic* pre-defined theory of PVS is used. Since rem(6)(0) = 0 is found true, the *assert* command which uses decision procedures to assert sequent formulas is able to prove the given goal. PVS can prove automatically many results through its automation. Therefore, when dealing with more complex problems, the user has the ability to concentrate on directing the steps to be taken rather than the focusing on other details.

4. Concerns

PVS does not support operator-defined theories, proof by contradiction, or derived rules. Operator theories are theories with operators that handle equations of associativity, commutativity, and identity efficiently. They can be important with respect to rewrite techniques. Proof by contradiction is an indirect method of proof where the proof assumes that the negation of a formula to be proved holds so that it can derive a contradiction. Derived rules are rules that can be used in addition to the basic proof rules, and each application of a derived rule can be replaced with a combination of applications of the basic proof rules. Adding derived rules may be necessary for conveniently embedding specification languages and their corresponding proof rules in the system, in order to provide a theorem proving environment for the original specification languages [Zha98]. PVS also lacks structures to support readability and ease of change. Therefore, a beginning user of PVS may find it difficult to create adequate and worthy specifications. Lastly, type checking is time critical since it is part of the interaction and can take up to thirty minutes to type check for large contexts [Kroening].

B. UNDERSTANDING THE PVS SPECIFICATIONS

1. The Formal Semantics of the PVS Specifications

a. The Simple Type Theory

i. Introduction

PVS is a strongly typed specification language. Expressions are checked to be well typed under a context that is a partial function that assigns either a TYPE, CONSTANT, or VARIABLE. Bool and real are base types that are examples of pretypes of the simple type theory. [A -> B] is defined as a function *pretype* from domain pretype A to range pretype B. A *type* is a pretype that has been typechecked in a given context.

The *preterms* of the language consist of the constants, variables, pairs, projections, applications, and abstractions. Lambda abstractions have the form LAMBDA(x: T), where T is a pretype and a is preterm.[Owr99]

ii. Contexts

A *context* is a sequence of declarations, where each declaration is either a type declaration s: TYPE, a constant declaration c: T where T is a type, or a variable declaration x: VAR T. Preterms and pretypes are typechecked with respect to a given context. [Owr99]

b. Subtypes

Subtyping is one of the main features of the PVS Specification Language. A *predicate type* in PVS is a function type where the range is the primitive type bool. A predicate is a term that has a predicate type. Since the elements of the subtype {x: T | a} satisfy the predicate λ(x: T): a, this is called a *predicate subtype* to distinguish it from other forms of subtyping. Type equivalence and type correctness are undecidable. Proof obligations are generated during typechecking and those obligations are the only source of such undecidability. [Owr99]

c. Dependent Types

In dependent typing, the type of one component of a product depends on the value of another component, or the type of the range of a function varies according to its argument value. An important degree of flexibility and precision is added to the type system when using dependent typing. Dependent typing representation can be seen below [Owr99].

- [i: nat, {j:nat | j ≤ i}]
- [i: nat, {j:nat | j ≤ i} -> bool]

2. Language of PVS Specifications

A PVS specification consists of a collection of theories. Each theory consists of a signature for the type names and constants introduced in the theory, and the axioms, definitions, and theorems associated with the signature.

a. Declarations

Entities of PVS are introduced by means of declarations that introduce types, variables, constants, formulas, judgements, and conversions. Declarations introduced in one theory may be referenced in another by means of the IMPORTING clause.

i. Uninterpreted Type Declarations: T: TYPE

Uninterpreted types support abstraction by providing a means of introducing a type with a minimum of assumptions on the type. TYPE+ signifies that the entity is nonempty.

ii. Uninterpreted Subtype Declarations: S: TYPE FROM T

K: TYPE FROM T has the same meaning as:

k_pred: [t -> bool]

k: TYPE = (k_pred)

iii. Interpreted Type Declarations: T: TYPE = nat

Interpreted type declarations are primarily a means for providing names for type expressions.

iv. Enumeration Type Declarations: T: TYPE = {a, b, c}; Enumeration type declarations are of the form: enumeration: TYPE = {e_1,…, e_n} where the e_i are distinct identifiers.

v. Empty vs. Nonempty Types

When type checking, the following rules are hold:

- Constants declared must be of a nonempty type
- Uninterpreted type or subtype declarations defined by TYPE can be empty
- TYPE+ is used for assuming nonemptiness for uninterpreted type declarations.
- TYPE+ is used for assuming nonemptiness for uninterpreted subtype declarations as long as the supertype is nonempty.

The type of an interpreted constant is nonempty, as the definition provides a witness and interpreted type declarations defined by TYPE are not assumed to be nonempty. An interpreted subtype declaration with a CONTAINING clause is considered nonempty. There is no TCC generated since the CONTAINING clause is a witness to the type.

vi. Variable Declarations

Variable declarations introduce new variables and associate a type with them. So that binding expressions and formulas can be succinct, these variable declarations provide a name an associated type.

vii. Constant/Fixed Declarations

PVS' underlying logic is higher order and the term constant refers to functions and relations as well as the usual constants.

viii. Formula Declarations

Axioms, assumptions, theorems, and obligations are formula declarations that are introduced with the keywords AXIOM, ASSUMPTION, THEOREM, AND OBLIGATION respectively. Axioms are not expected to have an associated proof and are assumed to be true.

b. TYPES

i. Subtypes

A subtype is any collection of elements of a given type itself forms a type. The supertype is the type from which elements are taken. The elements that form the subtype are determined by a subtype predicate on the supertype. Much of the expressive power of the language comes from the subtypes in PVS, however the cost is making typechecking undecidable.[Owr01*] When typechecking is undecidable, this leads to proof obligations (TCCs). These proof obligations can be discharged with the assistance of the PVS prover. The TCCs provide a debugging mechanism which point out flaws in the specification or what can be added for improvement. In the end, the TCCs generated provide a helpful tool to potential users when trying to build a specification that best represents their model.

ii. Function Types

Function Types come in the form: X: x -> y

iii. Record Types

Record types have the form [# a_1 : t_1, …, a_n : t_n #].

c. Expressions

i. Boolean Expressions

ii. IF-THEN-ELSE

iii. Binding Expressions

Binding expressions use keywords such as FORALL, EXISTS, or LAMBDA and consist of operations, a list of bindings, and an expression.

iv. Set Expressions

Sets of elements of type t are represented as predicates in PVS. For example, functions from t to real.

v. Record Accessors

If x is of type [# k: int, l: nat #], an x component can be accessed by x`k or k(r).

vi. COND Expressions: The following expressions are the same.

```
COND
  A_1 -> B_1
  A_2 -> B_2
  …
  ELSE -> B_N
ENDCOND
```

```
 IF A_1 -> B_1
ELSIF A_2 THEN B_2
…
ELSEIF A_N-1 THEN B_N-1
ELSE B_N
```

d. Theories

i. Theory Parameters

Theory parameters can consist of types, subtypes, constants, and imported theories. When instantiating a theory within another theory, the specification needs to provide actual parameters that substitute for the formals.

PVS Language	Example in Specifications
Uninterpreted Type Declaration	*State: Subject: TYPE+*
Uninterpreted Subtype declaration	*FTLSSpec: ProcessID: TYPE+ FROM nat*
Interpreted Type Declaration	*State: Access: TYPE = [# u: Subject, f: Object, m: Mode #], Access_State: TYPE = setof[Access]*
Enumeration Type Declaration	*State: Mode: TYPE = {rd, wr}*
Empty vs. Nonempty Types	*FTLSSpec: MemBlock: TYPE+ = [# lb: Memory, ub: {x: Memory | x >= lb} #] CONTAINING (# lb := MAXMEM, ub := MAXMEM #), ProcessID: TYPE+ FROM nat*
Variable Declarations	*FTLSSpec: p: var Processes*
Constant Declarations	*State: addit, del: Action, no_op: setof[Action]*
Formula Declarations	*FTLSSpec: AdditNotDel: AXIOM NOT addit = del*
Subtypes	*FTLSSpec:* *SystemMemBlocks: TYPE+ = {SetofMB | ValidMemBlocks(sysmb)}CONTAINING {x: MemBlock | x = (# lb := MAXMEM, ub := MAXMEM #)}*
Function Types	*State: slSubject: Subject -> Label*
Record Types	*State: Access: TYPE = [# u: Subject, f: Object, m: Mode #]*
Boolean Expression	*State: AddnotDel: bool = NOT addit = del*
IF-THEN-ELSE	*FTLSSpec: SecureProcTransform(r: Request, Pr: Processes): Processes = IF SecureRequest?(r) THEN ProcTransform(r, Pr) ELSE Pr ENDIF*
Binding Expressions	*FTLSSpec: ValidMemBlocks(smb: setof[MemBlock]): bool = FORALL(e, b: MemBlock) : member(e, smb) AND member(b, smb) => (e`lb > b`ub) OR (b`lb > e`ub) AND NOT e = b*
Set Expressions	*FTLSSpec: SetofMB: TYPE+ = setof[MemBlock]CONTAINING* *{x: MemBlock | x = (# lb := MAXMEM, ub := MAXMEM #)}*
Record Accessors	*FTLSSpec: ValidMemBlocks(smb: SetofMB): bool =* *FORALL (e, b: MemBlock):* *NOT e = b AND member(e, smb) AND member(b, smb) =>* *(e`lb > b`ub) OR (b`lb > e`ub)*

Table 5. Examples of the Language of PVS

C. ANATOMY OF THE FORMAL SECURITY POLICY MODEL SPECIFICATION

1. Elements

a. Subjects

Subject: TYPE+

b. Objects

Object: TYPE+

c. Access Attributes {read, write}

Mode: TYPE+, rd: Mode, wr: Mode

d. Security Levels (Top Secret, Secret, Classified, Unclassified)

See specification SLabels

2. Components

a. Current Access set

The current access set defines the access state as a set of triples (subject, object, access-attribute) where the "subject" has current "attribute" access to "object". It does not identify all the possible accesses, but only identifies one possible state which is the one the system is in currently. In the *State* specification, an instance of *Access_State* defines a current access set.

Access: TYPE = [# u: Subject, f: Object, m: Mode #]

Access_State: TYPE = setof[Access]

b. Object hierarchy

This consists of a parent-child relation structure on objects where the security level of the parent dominates the security level of the child. This hierarchy will not be given importance and therefore will not be represented in our formal security policy model.

c. Security Level function

The security level function determines the security levels for subjects and objects.

slSubject: [Subject -> Label]

slObject: [Object -> Label]

slSubject and *slObject* are functions that take a Subject and Object(respectively) and return an associated fixed security label of that subject and object(respectively). Each subject/object cannot have more than one security label associated with it and the security label associations are fixed.

d. Access matrix

The access matrix used in this paper is a simple representation of subject/object accesses. As seen in Table 4, one column is for each object (including subjects that are objects), and one row is for each subject. Each cell contains sets of access attributes. The cell of the ith row and the jth column contains the access attributes of the ith subject in the matrix (S_j) to the jth object in the matrix, (O_j). The current access set is a subset of the access matrix.

```
SecureAccess(a: Access): bool =
   COND a`m = rd -> slSubject(a`u) >= slObject(a`f),
   a`m = wr -> slSubject(a`u) = slObject(a`f)
   ENDCOND

SecState(st: Access_State): bool =
FORALL (a: Access): member(a, st) -> SecureAccess(a)


Access_matrix:  {a: Access | SecureAccess(a) }
```

SecureAccess is a boolean function that defines whether a given access relation is secure. Therefore, it checks to make sure that read and write modes are only assigned when the properties of our security policy are satisfied. This is achieved through a security label comparison between the subject and object (*slSubject* and *slObject* respectively). *Access_matrix* is the set of accesses that are secure and represents the super matrix. Since it is not needed in the specification, it has not been included in our *State* specification. *Access_State* and *Sec_State* are subsets of *Access_Matrix*. *SecState* and *SecureAccess* together represent the security policy.

3. Properties

The system should satisfy Property 1, 2, and 3 from above

4. Rules

a. State transition operators

- Altering current access
  - Get access (add to the current access set)
  - Release access (remove from the current access set)

```
Transform(e: TransformInstance, st: Access_State): Access_State =
   COND e`ac = addit AND SecureAccess(e`a) -> add(e`a, st),
            e`ac = del AND st(e`a) -> remove(e`a, st),
           member(e`ac, no_op) -> st,
      ELSE -> st
   ENDCOND
```

In Transform, when adding an access to current access set, the access is first checked to be secure, and then is added to the set of current accesses. When removing an access from the current access set, the access first must be in the set of all allowed accesses, and is then removed from that state. Otherwise, if the action of altering current access is neither adding to or removing from the current access set, then the access state(set of all accesses) remains the same. A transform function takes a current access set (*Access_State*) and a *TransformInstance*(specifying the transformation operation and the access involved) and returns a new current access set.

D. CONSTRUCTION OF THE FORMAL TOP LEVEL SPECIFICATION IN PVS

1. Anatomy of our model

a. Elements

i. Memory Blocks

Memory is a nonempty set of natural numbers that is bounded by some constant natural number represented as *MAXMEM*. A memory block is a nonempty entity represented as *MemBlock*, a record whose elements consist of the lower and upper bound values(located in memory) of the memory block. System memory blocks are represented as a set of valid memory blocks(non-overlapping memory blocks). System memory is then defined as a fixed set of system memory blocks. *MemBlockInstance* is a record whose elements consist of a memory block located in system memory and its associated access privilege.

```
MAXMEM: nat
Memory: TYPE+ = {mem: nat | mem <= MAXMEM}
CONTAINING MAXMEM

MemBlock: TYPE+ = [# lb: Memory, ub: {x: Memory | x >= lb}
#] CONTAINING (# lb := MAXMEM, ub := MAXMEM  #)

SetofMB: TYPE+ = setof[MemBlock] CONTAINING
{x: MemBlock | x = (# lb := MAXMEM, ub := MAXMEM #)}

ValidMemBlocks(smb: SetofMB) bool = FORALL (e, b:
MemBlock):
NOT e = b AND member(e, smb) AND member(b, smb) => (e`lb >
b`ub) OR (b`lb > e`ub)

SystemMemBlocks: TYPE+ = {sysmb: SetofMB |
ValidMemBlocks(sysmb)}CONTAINING {x: MemBlock | x = (# lb
:= MAXMEM, ub := MAXMEM #)}

SysMem: SystemMemBlocks

MemBlockInstance: TYPE = [# mb : {x: MemBlock | member(x,
SysMem)}, m: Mode #]
```

ii. Processes (Identified through a Process Identification Number)

ProcessID is an element from the nonempty set of natural numbers. A process is associated with a set of memory blocks each with a given mode, as well as a process identification number. Therefore, a process is represented as an entity that takes a ProcessID number and returns a set of memory blocks associated with that process identification number. It is not a function since one process can have different sets of memory blocks associated with it at various times.

```
ProcessID: TYPE+ FROM nat

ProcessMB: TYPE =  setof[MemBlockInstance]

Processes: TYPE = [ProcessID -> ProcessMB]
```

iii. Access Attributes {read, write}

Mode: TYPE+

rd: Mo de

wr: Mode

iv. Security Levels (Top Secret, Secret, Classified, Unclassified)

Refer to the *Slabels* specification

b. Components

i. Mapping to the Current Access Set

The current access set in the formal security policy model was an instance of Access_State. Process_State in the *FTLSSpec,* our specification representing the FTLS, provides a direct mapping from the FTLS to the current access set. It defines a function that takes a variable of type Processes and returns an Access_State where the elements of the FTLS can be mapped to elements of the Access_State.

slMemBlock: [MemBlock -> Label]

slPr: [ProcessID -> Label]

p: VAR processes

Process_State(p): Access_State = {a: Access | EXISTS (mbi: MemBlockInstance, PID: ProcessID): a`u = PID AND a`f = mbi`mb AND a`m = mbi`m AND member(mbi, p(PID))}

ii. Mapping to the Access matrix and the Security Policy

The comparison of security labels generated through the functions *slPr* and *slMemBlock* of a process and a memory block determine read and write privileges as stated in *SecProcess*. This Boolean function states that if all the memory blocks associated with a specific process satisfy the security requirements of the policy, then that process is secure. *SecProcesses* states that if all processes are secure, then the current state is secure. This could be represented as a matrix of memory blocks to *ProcessID*'s where the cells contain the access privileges.

```
SecProcess(p: Processes, pid: ProcessID): bool =  FORALL (mbi:
MemBlockInstance | member(mbi, p(pid))):
  COND mbi`m = rd -> slPr(pid) >= slMemBlock(mbi`mb),
      mbi`m = wr -> slPr(pid) = slMemBlock(mbi`mb)
  ENDCOND
SecProcesses(pr: Processes): bool =
     FORALL (pid: ProcessID): SecProcess(pr, pid)
```

iii. Level function

slMemBlock: [MemBlock -> Label]

slPr: [ProcessID -> Label]

slMemBlock and slPr are two functions that determine the security labels of memory blocks and processes respectively.

c. Properties

- A process has read access to a memory block only if its security label is greater than or equal to the security label of the memory block
- A process has write access to a memory block only if its security label is equal to the security label of the memory block

```
SecProcess(p: Processes, pid: ProcessID): bool =  FORALL (mbi:
MemBlockInstance | member(mbi, p(pid))):
  COND mbi`m = rd -> slPr(pid) >= slMemBlock(mbi`mb),
      mbi`m = wr -> slPr(pid) = slMemBlock(mbi`mb)
  ENDCOND
```

d. Rules

- State transition operators
  - Altering process table
    - Add memory block to a process
    - Remove a memory block from a process

If a request's action is adding a memory block, then add that request's memory block to the request's process. However, if a request's action is deleting a memory block, then remove that request's memory block from the request's process.

```
ProcTransform(r: Request, Pr: Processes): Processes =
  COND ac(r) = addit AND SecureRequest?(r) ->
     LAMBDA (x: ProcessID):
      IF NOT x = r`p THEN Pr(x) ELSE add(r`mbi, Pr(x)) ENDIF,
    r`ac = del ->
     LAMBDA (x: ProcessID):
      IF NOT x = r`p THEN Pr(x) ELSE remove(r`mbi, Pr(x)) ENDIF,
    ELSE -> Pr
  ENDCOND

  SecureRequest?(r: Request): bool =
    COND r`ac = addit ->
       ((r`mbi`m = rd AND slPr(r`p) >= slMemBlock(r`mbi`mb)) OR
        (r`mbi`m = wr AND slPr(r`p) = slMemBlock(r`mbi`mb))),
      ELSE -> TRUE
    ENDCOND
```

E. IMPORTANT NOTES

1. *Sec_theory* provides the Basic Security Theorem

The theorem that gets proven at the model level is *sec_transform*. This theorem's overall meaning is that if that if the initial state of the system is secure, then it will never enter a non-secure state given the operations (transforms) as defined. The model must represent the initial state of a system, the way in which the system progresses from one state to another, and a definition of a "secure" state of the system. *Sec_theory* is an abstract representation of the Basic Security Theorem. By mapping *State* and *FTLSSpec* to *Sec_theory*, this verifies that both these specifications satisfy the Basic Security Theorem properties given their different defined states.

2. Prelude File of PVS

Specifications for many foundational and standard theories are preloaded into PVS as prelude theories and are always available and do not need to be explicitly imported. In our specifications we use the function *add(x, a)*, *remove(x, a)*, *member(x, a)*, and *setof(x)* which are already defined functions in the *sets* and *defined_types* theory. The functions are defined as follows from the prelude:

- add(x, a): (nonempty?) = {y| x=y OR member(y, a)}
- remove(x, a): set = {y| x/= y AND member(y, a)} , /= represents not equals
- member(x, a): bool = a(x)
- setof: TYPE = [t -> bool] where set: TYPE = setof[t]

3. Disjointness in our Specifications

In the *State* specification, the disjointness of memory blocks and their read/write constraints is implicit, however in the *FTLSSpec* it becomes explicit.

4. Bus Error

A few times during the construction of our specifications, we would receive a message stated “error signal bus” which did not allow us to work with our specifications. To fix this problem, we had to delete all the files in the directory we were using except for the specification files. Since the proof files were removed, we had to perform all the proofs again. Since our specifications were kept simple, the proof construction was not too time consuming. However, in cases of larger applications, this can be great problem especially under time constraints.

F. INTER-LEVEL MAPPING OF THE FTLS AND SECURITY POLICY MODEL USING PVS

1. The Importing Function

When using the *IMPORTING* clause, the actual parameters provided are known as a *theory instance*. An *IMPORTING* clause forms a relation between the theory containing the *IMPORTING* and the theory referenced. If the entities are visible in the *IMPORTING* clause at some point in the theory, then they are visible to every declaration following. The *IMPORTING* clause can be used to map one specification to another specification [Owr01]. The State theory is imported into the FTLS specification using the IMPORTING clause as follows (See Figure 6):

<u>State</u>
State[Subject: TYPE+, Object: TYPE+, (IMPORTING SLabels) slSubject: [Subject -> Label], slObject: [Object -> Label], Action: TYPE+, addit: Action, del: Action, no_op: setof[Action], Mode: TYPE+, rd: Mode, wr: Mode]: THEORY

<u>*FTLSSpec*</u>
IMPORTING State[ProcessID, MemBlock, slPr, slMemBlock, Action, addit, del, no_ops, Mode, rd, wr]

2. Correspondence between State Elements of the FTLS and State Elements of the Formal Security Policy Model

A *ProcessID* that identifies a process in the FTLS becomes a *Subject* in the security policy model. A memory block represented as *MemBlock* in the FTLS becomes an *Object* in the security policy model. As stated in Chapter V, Section D, *Process_State* in the FTLS corresponds to the current access set of the formal security policy model. The security level functions and *SecProcesses* of the FTLS correspond to the set of all allowed accesses in the *State* specification,

3. Correspondence between Transform Functions of FTLS and Transform Functions of the Formal Security Policy Model

Addition of a memory block to a process (*addit*) and a removal of a memory block from a process (*del*) in the FTLS, becomes *addit* and *del* respectively in the security policy model. The action *addit* in the security policy model is the addition of an access to the current access set, and *del* is the removal of an access from the current access set. *ProcTransform* is the function that adds or removes memory blocks from a process and therefore behaves similarly to *Transform* in the *State* specification.

4. Code Correspondence

When implementing a security kernel, it is essential to keep the number of lines of code to a minimum. This offers verifiable protection and allows the kernel to be analyzed more efficiently. A minimized computer system is a system that only implements the security-related features or requirements of the policy. Therefore, by limiting the amount of code and mapping the implementation to the FTLS to the formal security policy model, a proof of correspondence from the code to the policy can be developed. Minimized systems provide much higher confidence against subversion. By having a proof of correspondence, untended functionality and bugs can be recognized and fixed in the implementation.

5. Functional Languages

When a functional specification is formal, the proof of correspondence shall also be formal [Com03]. A functional style formal language's only representation for an operation is a function and there are only expressions and not statements. A functional language is useful when writing specifications of models since it has a considerable body of theory for automatically reasoning about functionally-expressed algorithms [Irv03]. Also, the combination of a functional programming language and theorem provers is much more natural than the combination of sequential programming languages and theorem provers. Thus, there are theorem provers that accept functional programming languages as input [Kro02]. Future research will be conducted to explain or "interpret" as to the approach for using a "functional" language for state machine modeling.

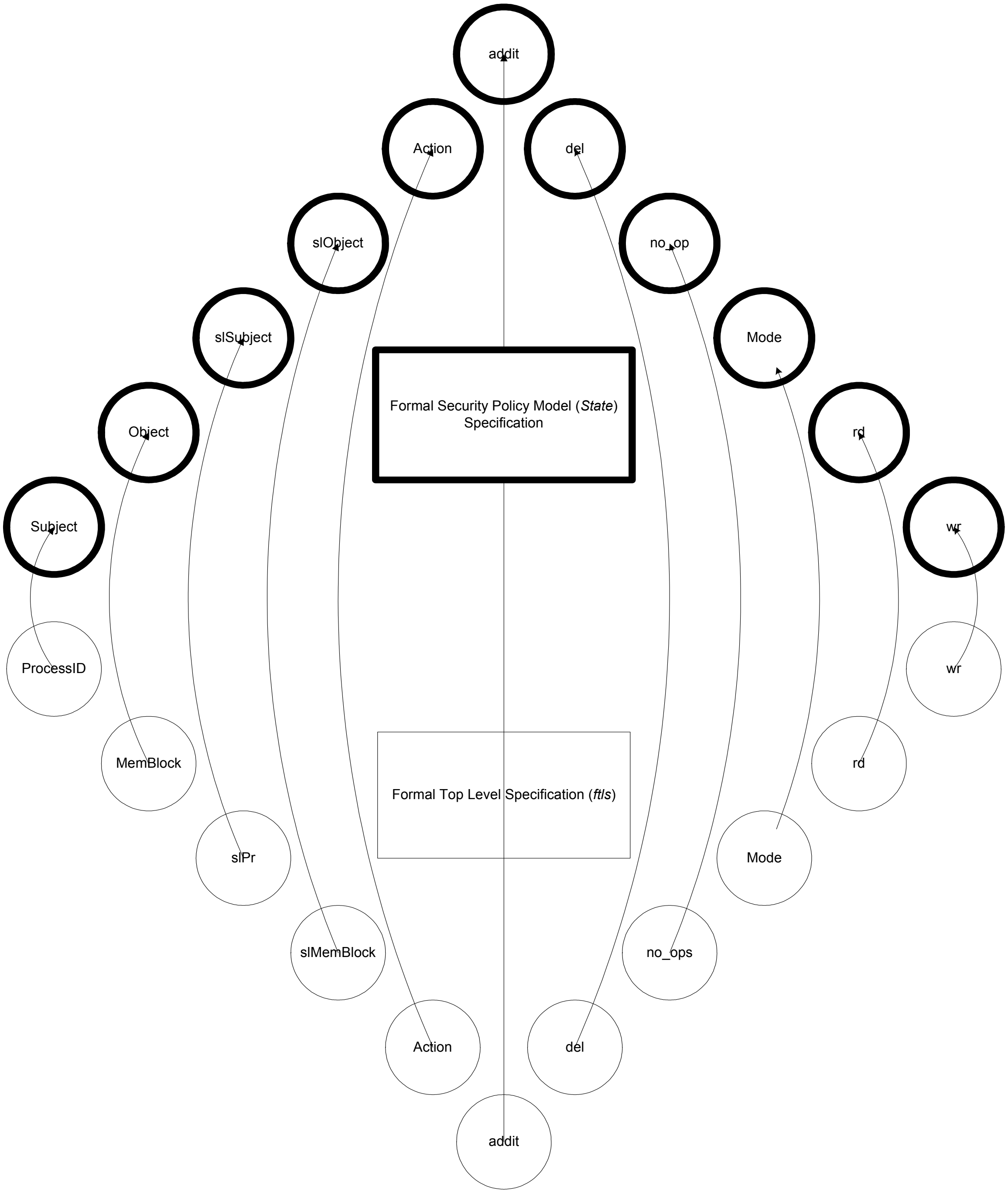


Figure 6. Inter-level Mapping

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS

PVS can be used to specify and validate system requirements, verify that an implementation meets the requirements and then help to refine the design in an effort to improve system performance. A specification is formed by combining theories describing various components and properties. Each theory is partitioned into assumptions, definitions, axioms, and theorems. Benefits of formal specifications are that they provide a higher level of rigor which enables a better understanding of the problem, defects are uncovered that would likely go unnoticed with traditional specification methods, identify defects earlier in the life cycle. Formal specifications enable formal proofs that can establish fundamental system properties as invariants. They encourage an abstract view of a system that focuses on what a proposed system should accomplish as opposed to how to accomplish it.

Verification can be achieved by identifying correctness conditions and using PVS to prove those correctness conditions. PVS has three features that allow it to detect deviations from these correctness conditions[Fre02].

- It is based on a typed higher-order logic
- It supports specification using a form of conservative extension
- The theorem prover provides a powerful, extensible system for verifying obligations

The process of adding definitions to an existing theory is called extension and is a common mechanism for writing specifications. A theory is a conservative extension of another if the extension adds axioms that only define properties over new operations. Therefore, if a consistent theory is extended in a conservative manner, the resulting theory is guaranteed to be consistent [Fre02]. In our specifications, we did not extend any existing theories, however this feature is of great importance when trying to ensure consistency of specifications.

Our methodology consisted of constructing specifications representing the formal security policy model and the formal top level specification, mapping the FTLS to the formal security policy model, and proving that the FTLS maps to the formal security policy model. The main purpose of developing these specifications was to demonstrate that PVS could support inter-level mapping between the FTLS and the formal security policy model. Based on our survey, PVS satisfied the evaluation criteria as seen in Table 6 that was developed prior to beginning constructing our specifications. The information used to establish our results in the survey was taken from available relevant documentation about the verification systems. This included the homepages of each of the verification tools, formal methods workshop summaries, and papers on projects that used verification tools. The list of evaluation criteria below represent criteria that could only be analyzed properly by developing our own specifications, providing inter-level mapping, and proving the necessary theorems.

- User Interface
- Consistency of Specifications
- User-Friendly
- User Presentation Language
- Multiple Levels of Abstraction
- Expressiveness

In the next section, how well PVS represented both formal security policy model and the FTLS will be analyzed based on the criteria seen above. From there, an analysis of how easy it was to prove that the FTLS mapped to the formal security policy model will be conducted. The last section will answer the question of what is the next step in our empirical study. It introduces ACL2, the next verification tool to be studied, gives a brief background and compares ACL2 to PVS, and lastly presents the methodology for constructing our specifications (i.e. the functions and theorems required).

A. EVALUATION OF HOW WELL PVS DESCRIBED THE FORMAL SECURITY POLICY MODEL AND THE FTLS

1. Expressiveness and User Presentation Language (Specification Language)

The specification language of PVS is based on classical, simply typed higher-order logic, but the type system has been augmented with subtypes and dependent types. Higher order logic provided a straightforward rigorous formalization of any mathematical definitions. One appeal of type theory is that the rendering of higher-level mathematical notions like functions is rather direct, whereas in set theory it relies on a few layers of definition. Our specifications could be written concisely in this high level language. The use of higher order logic and the type system allowed us to closely represent the original requirements of the policy and an abstract of the implementation. The expressiveness of the PVS specification allowed us to express the formal security policy model and the FTLS succinctly. The use of PVS's type system and its underlying logic allowed gave us an important degree of flexibility and precision when constructing our specifications.

a. Predicate Subtypes and TCCs

Predicate subtypes provide a mechanism for defining new types using comprehension. Given any predicate, s?, with domain, D, the predicate subtype(s?) is defined as {d: D | s?(d)} [Fre02]. When we used predicate subtypes, it was possible for the type checking system of PVS to automatically generate and verify the obligations. Predicate subtypes can also be used to check statically for violations such as division by zero or out-of-bounds array references, and can also express more sophisticated consistency requirements. They are also used to constrain domain/range of operations and to define partial functions. PVS does not make any assumptions about the cardinality of the sets that interpret its types. Therefore, sets can be empty, finite, or infinite. When a predicate subtype is used, the cardinality cannot be checked algorithmically and therefore an "existence TCC" is generated. This "existence TCC" is a potent detector of erroneous specifications when higher (i.e. function and predicate) types are involved [Rus98]. Therefore, in our specifications we added the CONTAINING clause to explicitly state that a specific set of elements was nonempty. After adding this clause, the proof obligations could be proved automatically.

b. Dependent Types

In PVS, function, tuple, and record types may be dependent in the sense that some of the type components depend on earlier components. This can be important when trying to prove a type check condition which can be seen in Part 2, *The Importance* *of TCCs*. In this example, by adding the statement that specifies that if *x* is a memory block element of *MemBlockInstance* then it is a member of system memory.

MemBlockInstance: TYPE = [# mb : {x: MemBlock | member(x, SysMem)}, m: Mode #]

As stated before, dependent typing increases the amount of flexibility and increases the precision of the specification. By using dependent typing, expressions can be more succinct and easier to understand.

c. Higher-Order Logic

Higher-order logic is a logical system, usually a Type Theory, with multiple ranges of quantification (usually called *types*) some of which contain sets or functions. Higher-order logic ensures that the specification language applies to the widest range of applications. In PVS, predicates and sets can be regarded as essentially equivalent. All members of a set are of the same type in higher-order logic. Therefore, PVS also allows set notation for predicates. The inclusion of lambda-abstractions allows the definition of unnamed functions as is traditional in many languages. Most provers support quantifiers, and a few go further. A higher-order syntax allows users to define new variable-binding constructs such as least n P (n). Since PVS is based on higher-order logic, we were able to write our specifications concisely. We were able to quantify over predicates or properties in our specifications. In our specifications we were able to use &, OR, =>, FORALL, EXISTS, etc..

A specification may be easier to code in higher-order logic, however, the proofs can become more complex. In our specifications, since our specifications were kept simple, our proofs were not complicated.

2. Importance of The Type Checking System

Type Checking Conditions (TCCs) are produced when typechecking a specification by checking the semantic constraints, determining the types of expressions, and resolving names. Strategies are used to basically expand all definitions of the TCC and add semantic information to the internal representation of the parser. Theorem proving may be required to establish the type-consistency of a PVS specification and these theorems that need to be proved are called TCCS. They can also be referred to as proof obligations.

The assumptions in the higher level specification become proof obligations (TCCs) in the lower level specification. For example, the assumptions in the *State* specification become TCCs in the *FTLSSpec* specification. When evaluating the computer system, the implementation should satisfy the assumptions stated in the FTLS. Since implementation details are not known currently, the assumptions are written as axioms in the *FTLSSpec* specification. Type checking conditions point out defects in the specification and demonstrate what needs to be added to or deleted from a specification. The PVS type checker can be considered as a debugger of the model.

a. Memory Block in System Memory

In the initial stages of developing our specification of the FTLS we assumed that a memory block that was to be added or deleted from a process was already in the system memory. However, proof obligations were generated from the definition of *ProcTransform.* We had to show that when adding or deleting a memory block from a process, the set of memory blocks associated with that process after the change would still be a subset of system memory. The TCCs generated were:

```
% Subtype TCC generated (at line 50, column 46) for add(r`m, Pr(x))
% expected type  ProcessMB unfinished
ProcTransform_TCC1: OBLIGATION FORALL (Pr: Processes, r:
Request):  r`act = addit IMPLIES  (FORALL (x: ProcessID): x = r`p
IMPLIES subset?[MemBlock](add[MemBlock](r`m, Pr(x)), SysMem));

% Subtype TCC generated (at line 53, column 46) for remove(r`m, Pr(x))
% expected type  ProcessMB unfinished
ProcTransform_TCC2: OBLIGATION FORALL (Pr: Processes, r:
Request):
r`act = del IMPLIES (FORALL (x: ProcessID): x = r`p IMPLIES
subset?[MemBlock](remove[MemBlock](r`m, Pr(x)), SysMem));
```

Therefore, we had to add that when requesting to add or delete a memory block from a process, the memory block itself would already be part of system memory to our specification. The following line changes were made:

```
MemBlockInstance: TYPE = [# mb : MemBlock, m: Mode #]
```

became

```
MemBlockInstance: TYPE = [# mb : {x: MemBlock | member(x, SysMem)}, m: Mode #]
```

b. Axiom Additions

When typechecking the *State* specification, the following TCCs were generated:

```
Disjointness TCC generated (at line 46, column 6) for
   % COND a`m = rd -> slSubject(a`u) >= slObject(a`f),
   %      a`m = wr -> slSubject(a`u) = slObject(a`f)
   % ENDCOND
  % proved - complete
SecureAccess_TCC1: OBLIGATION FORALL (a: Access): NOT (a`m = rd AND a`m = wr);

% Coverage TCC generated (at line 46, column 6) for
   % COND a`m = rd -> slSubject(a`u) >= slObject(a`f),
   %      a`m = wr -> slSubject(a`u) = slObject(a`f)
   % ENDCOND
  % proved - complete
SecureAccess_TCC2: OBLIGATION FORALL (a: Access): a`m = rd OR a`m = wr;

% Disjointness TCC generated (at line 54, column 6) for
   % COND e`act = addit AND SecureAccess(e`ac) -> add(e`ac, st),
   %      e`act = del AND st(e`ac) -> remove(e`ac, st),
   %      member(e`act, no_op) -> st,
   %      ELSE -> st
   % ENDCOND
  % proved - complete
Transform_TCC1: OBLIGATION
  FORALL (e: TransformInstance, st: Access_State):
   NOT ((e`act = addit AND SecureAccess(e`ac)) AND e`act = del AND st(e`ac))
    AND
    NOT ((e`act = addit AND SecureAccess(e`ac)) AND
          member[Action](e`act, no_op))
     AND NOT ((e`act = del AND st(e`ac)) AND member[Action](e`act, no_op));
```

Therefore, based on these TCCs, we had to explicitly write some assumptions that we had ignored but were of great importance in our model.

AddDelNotNo_Op: ASSUMPTION NOT member(addit, no_op) AND NOT member(del, no_op)

AddnotDel: ASSUMPTION NOT addit = del

RdnotWr: ASSUMPTION NOT rd = wr

AddDelNotNo_Op provides the assumption that the actions *addit* and *del* are not members of the set of actions that do not have any operations. *AddnotDel* and *RdnotWr* provide assumptions of disjointness that *addit* and *del* are not the same entities and read and write are not the same entities respectively. From typechecking the PVS theories, we discovered the need for axioms or assumptions that needed to be added to the specification. These then lead to properties of the FTLS and hence the implementation that we might not have realized if we had not developed the specifications.

PVS, in addition can provide counterexamples to original assumptions and thereby suggest extra constraints that need to be formalized [Kro02]. By constructing a PVS specification of our model, we have the advantage of using the PVS system to analyze the model. This PVS system allows us to ask questions about our model, such as the emergent properties of the model. These properties are not explicitly described via the axioms of the model itself; rather, they are logical consequences of the axioms [Pai01].

The type checker can detect a lot of common specification errors. Since it has to be performed again after any change to the theory, it is part of the interaction of developing a valid specification. It is the type checking system as well as the expressiveness of PVS that allowed us to write concise and suitable specifications for representing both the FTLS as well as the formal security policy model. The generation of proof obligations provided us with ways to improve the specifications and the specification language together with its underlying logic allowed us to write our specifications in a flexible and straightforward manner.

B. EVALUATION OF HOW EASY IT WAS TO PROVE THAT THE FTLS MAPPED TO THE FORMAL SECURITY POLICY MODEL

1. Decision and Inference Strategies

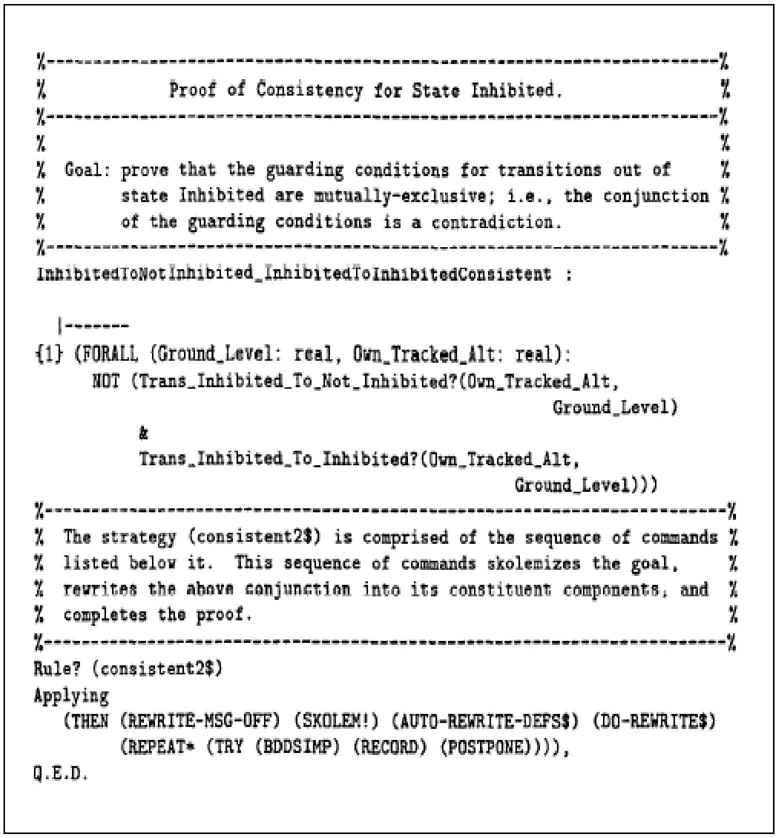
PVS provides a language for defining high-level inference strategies. The typical strategies include those for instantiation of quantifiers, repeated skolemization, simplification, rewriting, and induction. The use of these powerful inference steps allows us to define a small number of flexible strategies that suffice for productive proof construction [Rus96]. Strong decision procedures have a high degree of automation. For example, the "grind" command is like automatic proving, even though this feature works only for simple theorems. We still needed to guide it through the proof. It plays the role of a proof verifier, rather than prover, since it just verifies small proof steps, rather than generating the proof on its own. However, when proving TCCs in our specifications, the grind command was highly useful. The "grind" command combines various commands into one and allowed us to prove the proof obligations quickly. In addition, there were times when the strategy for the proof was not apparent, and using the grind command either simplified or automatically solved the proof. Since our specifications were kept simple to illustrate inter-level mapping, the proofs themselves (TCCs and theorems) were not difficult to prove. There were some instances however when only by attempting to use a specific command could we determine its applicability to solving the proof.

In a theorem prover, the same axiom can be used in many different ways, depending on the context. So a theorem prover is less predictable but more flexible than a programming language such as Prolog. When we are searching for contradictions, we don't know in advance how the axioms are to be executed, so we use a theorem prover. The "use" command invokes lemmas with instantiation. This allows us to use existing lemmas, axioms, theorems, etc from the PVS prelude file or our own defined assumptions, axioms, theorems, lemmas, etc. in our specification was used quite frequently. Using this command helped us prove many of the proof obligations from our specifications. An explanation of the various commands used in our proofs please refer to [Sha01] and the proofs can be seen in Appendix M through O.

2. Interface

a. Usability

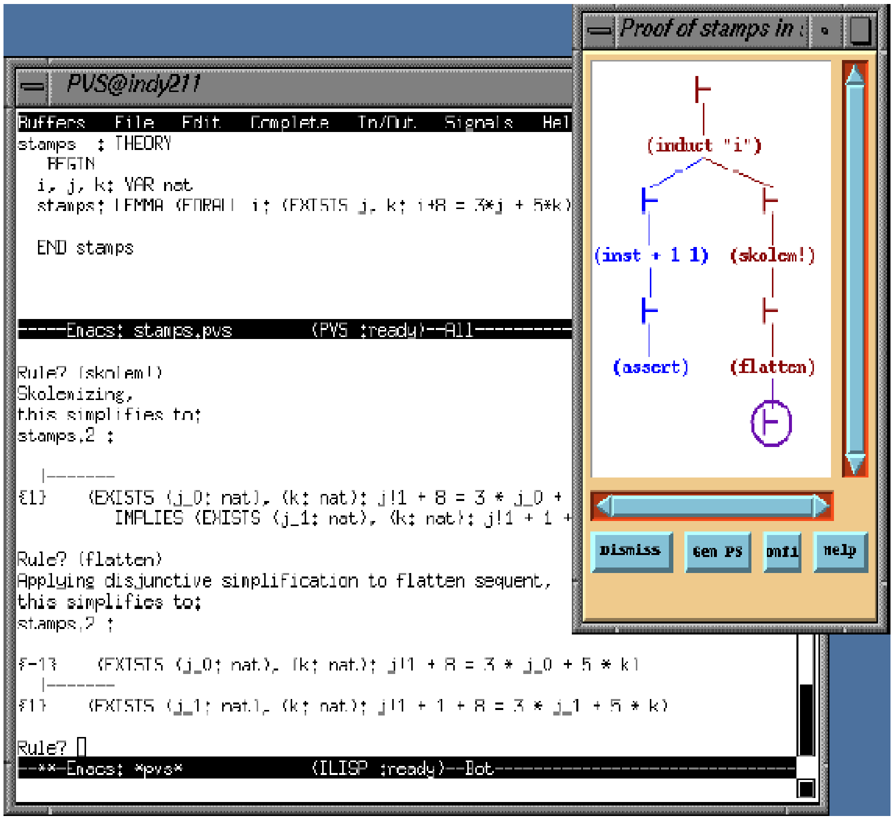
The usability of PVS for larger proofs relies on the existence of powerful procedures and strategies. Therefore, planning needs to be performed at a more intuitive level. A theorem prover should have a set of commands that maps well to the corresponding actions and is easy to understand and recall in order to facilitate plan formation. With automation taking care of the low level proof, the complexity of the built-in procedures means that the outcome of commands cannot be predicted and some experimentation is nearly always required [Mer96]. Since PVS's theorem prover is interactive, we essentially had to know the strategy we were going to use in our proof before we started. Otherwise, we could end up stuck in our proof construction by going down the wrong path. There were help buffers that allowed us to view either one or the whole range of possible commands and their correct usage. The interactive prover introduced flexibility and we were able to use our own strategies to perform the proofs. In other words, PVS supported "user-defined proof strategies". For example, Czerny and Heimdahl in "Using PVS to analyze hierarchical state-based requirements for completeness and consistency" define their own strategy called consistent2 to prove that their specifications are consistent seen in Figure 7.



```
%---------------------------------------------------------------------%
%             Proof of Consistency for State Inhibited.               %
%---------------------------------------------------------------------%
%                                                                     %
%  Goal: prove that the guarding conditions for transitions out of    %
%        state Inhibited are mutually-exclusive; i.e., the conjunction %
%        of the guarding conditions is a contradiction.               %
%---------------------------------------------------------------------%
InhibitedToNotInhibited_InhibitedToInhibitedConsistent :

  |-------
{1} (FORALL (Ground_Level: real, Own_Tracked_Alt: real):
      NOT (Trans_Inhibited_To_Not_Inhibited?(Own_Tracked_Alt,
                                              Ground_Level)
           &
           Trans_Inhibited_To_Inhibited?(Own_Tracked_Alt,
                                         Ground_Level)))
%----------------------------------------------------------------------%
%  The strategy (consistent2$) is comprised of the sequence of commands %
%  listed below it.  This sequence of commands skolemizes the goal,     %
%  rewrites the above conjunction into its constituent components, and %
%  completes the proof.                                                 %
%----------------------------------------------------------------------%
Rule? (consistent2$)
Applying
   (THEN (REWRITE-MSG-OFF) (SKOLEM!) (AUTO-REWRITE-DEFS$) (DO-REWRITE$)
        (REPEAT* (TRY (BDDSIMP) (RECORD) (POSTPONE)))),
Q.E.D.
```

Figure 7. User Defined Strategies in PVS

When constructing our proof, after each command is entered, the current sequent is displayed. Therefore, we were able to get an overview of our progress made at each command entered. It is also possible to view the current proof steps as formatted text where branching is indicated by indentation which allows the tree structure to be clear. This can be seen in the Figure 8 below [Mer96].



```
stamps  : THEORY
   BEGIN
  i, j, k: VAR nat
  stamps: LEMMA (FORALL i: (EXISTS j, k: i+8 = 3*j + 5*k)

  END stamps



-----Emacs: stamps.pvs          (PVS :ready)--All----------

Rule? (skolem!)
Skolemizing,
this simplifies to:
stamps.2 :

  |-------
{1}    (EXISTS (j_0: nat), (k: nat): j!1 + 8 = 3 * j_0 +
           IMPLIES (EXISTS (j_1: nat), (k: nat): j!1 + 1 +

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
stamps.2 :

{-1}    (EXISTS (j_0: nat), (k: nat): j!1 + 8 = 3 * j_0 + 5 * k)
  |-------
{1}    (EXISTS (j_1: nat), (k: nat): j!1 + 1 + 8 = 3 * j_1 + 5 * k)

Rule? 
--**-Emacs: *pvs*          (ILISP :ready)--Bot-----------------------------
```

Figure 8. Interface of PVS theorem prover

b. *Concerns with the Interface*

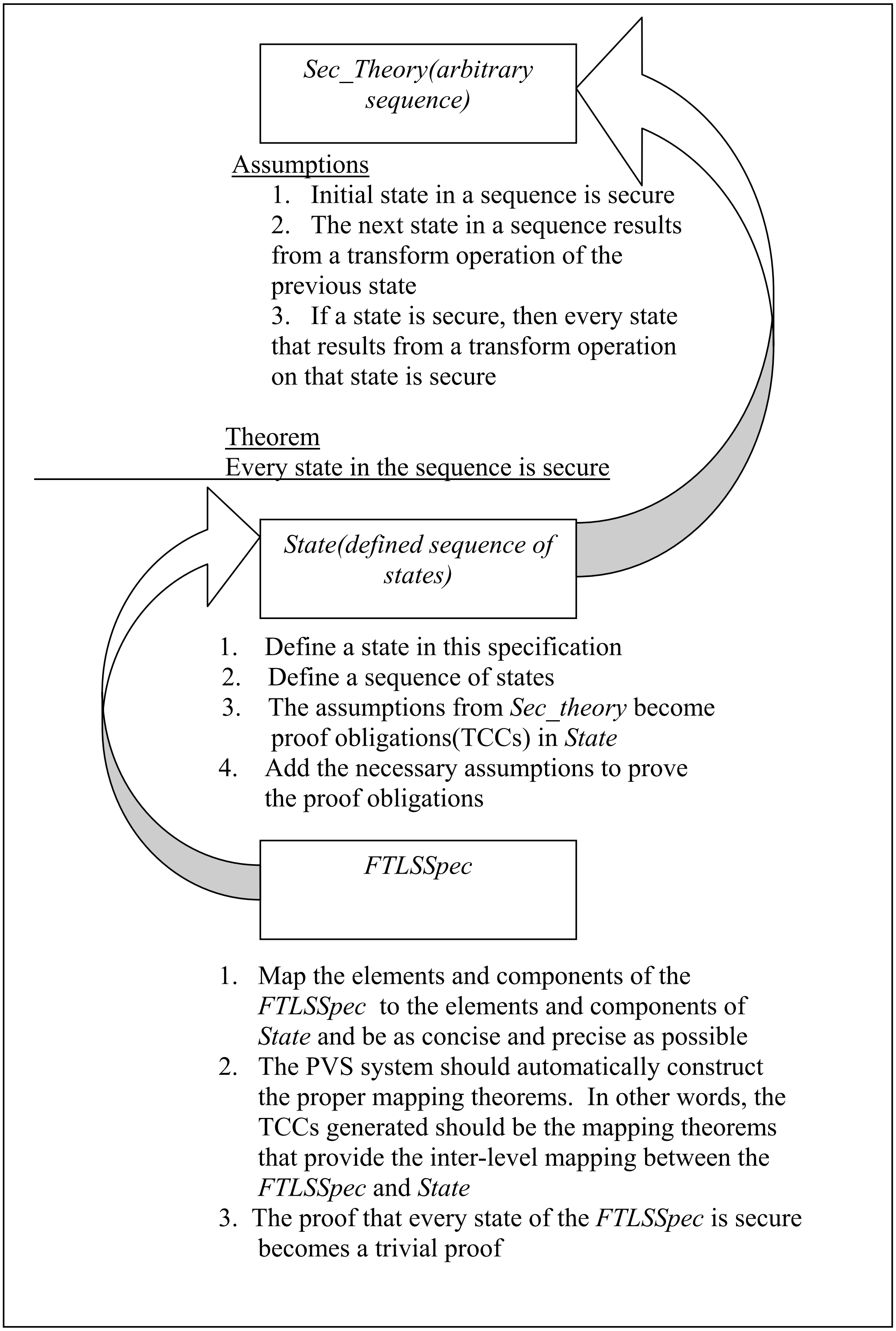
Since PVS cannot display standard mathematical symbols, ASCII alternatives are used instead. The method of best representing mathematics and logic is up to the user. "Pretty-printing" as the name states helps specifications become more organized and easier to read. Insertion carriage-returns and indentation are used to make the structure of the specification more readily apparent. However, how "pretty" the specification becomes is once again a decision made by the user. A problem with "pretty-printing" is that it removes comments made in the specification. So, even though it helps to structure the specification, it can make the specification harder to understand.

c. *Implementation Language*

When using the help buffers and viewing the theorem prover interface, the descriptions and messages are somewhat cryptic unless the user is familiar with the LISP programming language. Proof commands are in Lisp syntax, where the first term identifies the command, the second generally indicates those formulas in the sequent to which the command should be applied, and any required PVS text is enclosed in quotes. However, even though not all of us were familiar with the Lisp language, we still were able to understand the proof commands and the interface of the theorem prover. Therefore, the fact that PVS's implementation language is Common Lisp may not be a hindrance to users who are not familiar with the language.

3. The Inter-level Mapping Problem

In the specifications, *State* is mapped to *Sec_theory*. This mapping supports the Basic Security Theorem principles where every state in the sequence of states of the *State* specification is secure. The main goal however is to map the *FTLSSpec* specification to the *State* specification so that the following occurs (See Figure 9).


Sec_Theory(arbitrary sequence)
State(defined sequence of states)
FTLSSpec

Assumptions

1. Initial state in a sequence is secure
2. The next state in a sequence results from a transform operation of the previous state
3. If a state is secure, then every state that results from a transform operation on that state is secure

Theorem
Every state in the sequence is secure

1. Define a state in this specification
2. Define a sequence of states
3. The assumptions from *Sec_theory* become proof obligations(TCCs) in *State*
4. Add the necessary assumptions to prove the proof obligations

1. Map the elements and components of the *FTLSSpec* to the elements and components of *State* and be as concise and precise as possible
2. The PVS system should automatically construct the proper mapping theorems. In other words, the TCCs generated should be the mapping theorems that provide the inter-level mapping between the *FTLSSpec* and *State*
3. The proof that every state of the *FTLSSpec* is secure becomes a trivial proof

Figure 9. Ideal Inter-level Mapping Construction

a. Initial Version of the Specifications

When importing a higher level theory into a lower level theory using the IMPORTING clause in our specifications, type-checking conditions (TCCs) are generated. These TCCs were to make sure that the assumptions stated in the higher level theory are satisfied in the lower level theory. The assumptions of the higher level theory become proof obligations in the lower level theory as stated before. However, a problem arose in trying to prove those type-checking conditions in an earlier version of our specifications. In the earlier version, *System* was in between the *State* and *FTLSSpec* levels. The following parameter list of the theory *System* was used to try to map the *FTLSSpec* theory to the *System* theory which in turn needed to map to the *State* theory.

> *System[Subject: TYPE+, Object: TYPE+, (IMPORTING SLabels) slSubject:[Subject -> Label], slObject: [Object -> Label], Action: TYPE+, addit: Action, del: Action, no_op: setof[Action], Mode: TYPE+, rd: Mode, wr: Mode, (IMPORTING State[Subject, Object, slSubject, slObject, Action, addit, del, no_op, Mode, rd, wr]) fs: sequence[Access_State]]: THEORY*

To be able to prove the TCCs generated from importing the *State* theory, we needed to be able to use the assumptions in the *System* theory. These assumptions were the same as the assumptions of the current *State* theory (Appendix M). However, the problem was that we could not use the assumptions of our *System* theory since they were defined after the IMPORTING clause was introduced. The IMPORTING clause functions by only using the information prior to its use. Since PVS did not allow us to embed assumptions in the parameter list, and we wanted the format of the parameter to remain the same to provide inter-level mapping, we were unable to prove some of type checking conditions. Solutions to this problem are left for future work. The current version of the inter-level mappings between our specifications can be seen in Figure 10.

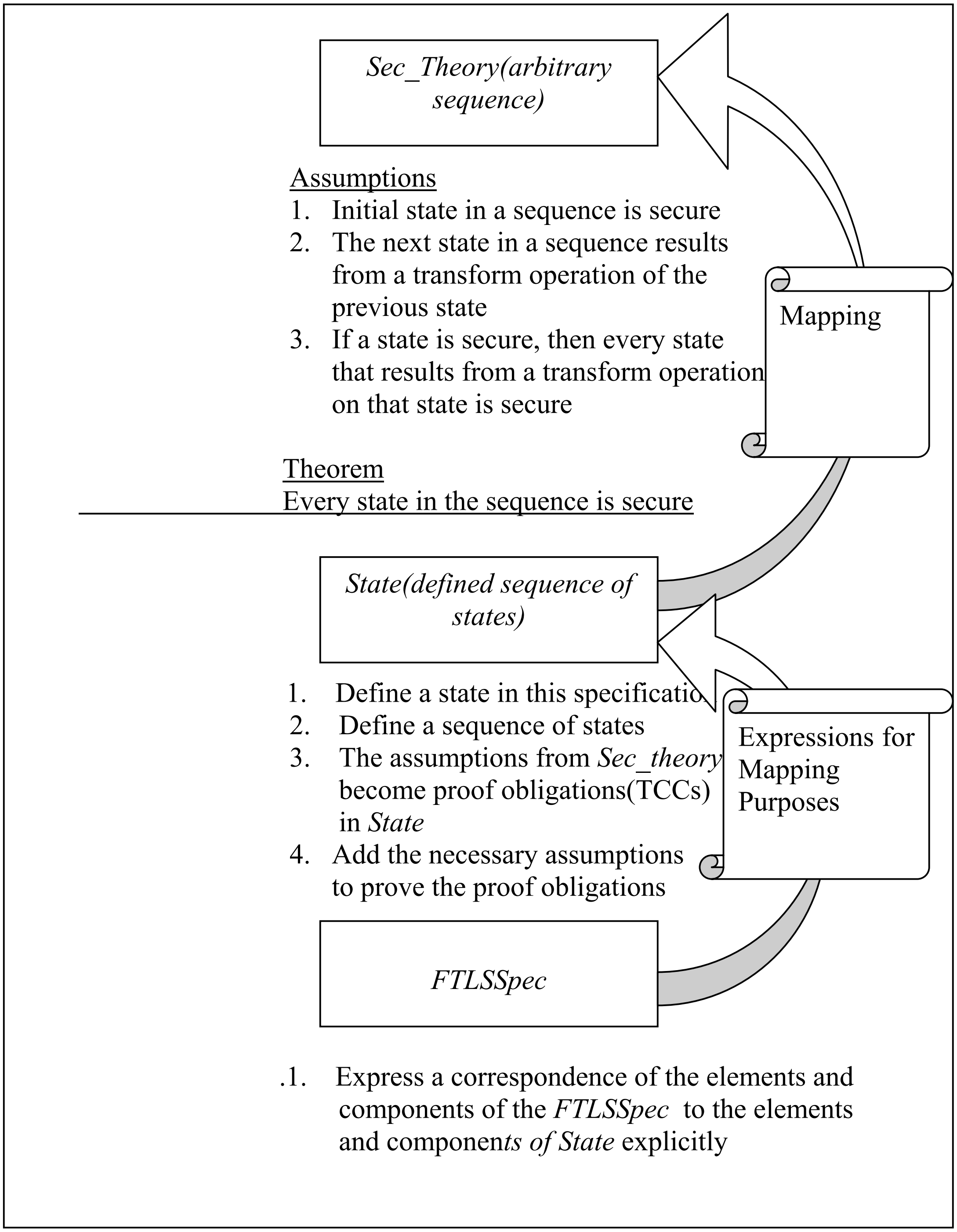

Sec_Theory(arbitrary sequence)
Assumptions
1. Initial state in a sequence is secure
2. The next state in a sequence results from a transform operation of the previous state
3. If a state is secure, then every state that results from a transform operation on that state is secure
Theorem
Every state in the sequence is secure
Mapping
State(defined sequence of states)
1. Define a state in this specificatio
2. Define a sequence of states
3. The assumptions from Sec_theory become proof obligations(TCCs) in State
4. Add the necessary assumptions to prove the proof obligations
Expressions for Mapping Purposes
FTLSSpec
.1. Express a correspondence of the elements and components of the FTLSSpec to the elements and components of State explicitly

Figure 10. Current Inter-level Mapping Construction

b. Mapping of Sec_theory and the State Specifications

Using the IMPORTING clause, we were able to map *State* to *Sec_theory* as seen in Figure 10. Proofs of the TCCs generated from this importation can be seen in

Appendix M. The IMPORTING clause was used to map *State* to *Sec_theory* as seen below. In the *State* specification, the names of the parameters remained the same, however the parameters were defined in *State.*

Sec_theory
Sec_theory[Access_State: TYPE, TransformInstance: TYPE, seq: sequence[Access_State], SecState:[Access_State -> bool], Transform: [TransformInstance, Access_State -> Access_State]]: THEORY

State
IMPORTING S ec_theory[Access_State, TransformInstance, seq, SecState, Transform]

c. Expressions for Mapping the FTLS Specification to the State specification

Since the system was unable to construct the necessary mapping theorems from our current specifications, we constructed components explicitly in *FTLSSpec* that corresponded with the components of the *State* specification. We were able to form a correspondence between the elements of the specifications using the IMPORTING clause as illustrated in Figure 6. Refer to Appendix M and N to view the specifications. Figure 11 is a diagram of the corresponding components of *FTLSSpec* and *State* as well as questions that arose when trying to provide inter-level mapping. Future research should be conducted so that the mapping between components is not explicitly stated and ideally the inter-level mapping should be performed as in Figure 9.

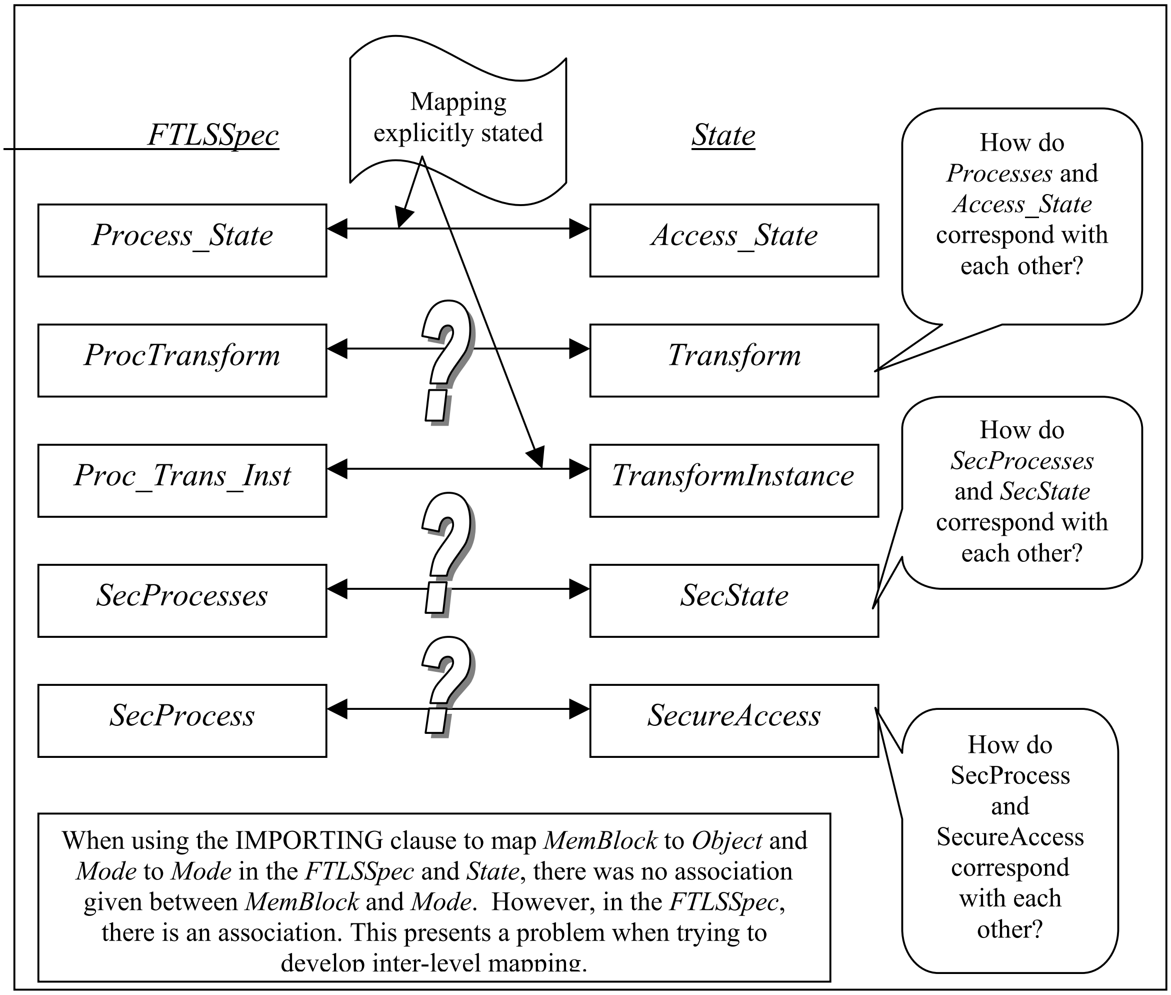

Mapping explicitly stated
FTLSSpec
State
How do *Processes* and *Access_State* correspond with each other?
Process_State
Access_State
ProcTransform
Transform
Proc_Trans_Inst
TransformInstance
How do *SecProcesses* and *SecState* correspond with each other?
SecProcesses
SecState
SecProcess
SecureAccess
How do SecProcess and SecureAccess correspond with each other?
When using the IMPORTING clause to map *MemBlock* to *Object* and *Mode* to *Mode* in the *FTLSSpec* and *State*, there was no association given between *MemBlock* and *Mode*. However, in the *FTLSSpec*, there is an association. This presents a problem when trying to develop inter-level mapping.

Figure 11. Inter-level Mapping Problem

Since most of our research time was spent on building the frameworks of the formal security policy model specification and the FTLS, the inter-level mapping could not be constructed since time was limited. However, whether this problem is based on the PVS system, the frameworks of our specifications, or inter-level mapping features of PVS that were unaware to us remains questionable. Since inter-level mapping between *FTLSSpec* and *State* could not be constructed, we were unable to analyze how easy it was to prove the mapping. Finding a solution to the inter-level mapping problem as well as analyzing how easy it is to prove that the FTLS mapped to the formal security policy model becomes a problem to answer in future work.

C. LEVEL OF ASSURANCE PROVIDED

Increasing the level of assurance that a system provides can be achieved by reducing requirements errors by applying formal methods during the requirements phase. A formal method is a development method based on some formalism, such as a formal specification notation or a formal analysis technique. A formal requirements specification can reduce errors by reducing ambiguity and imprecision and by making some instances of inconsistency and incompleteness obvious. Formal analysis can detect many classes of errors in requirements specifications, some of them automatically. Formal specifications as a system description clarify requirements and high-level design, articulate implicit assumptions, identify undocumented or unexpected assumptions, expose flaws, and identify exceptions. Error detection techniques can be seen below. PVS supports all of the features below except for providing executable specifications.

- Inspection of the formal specification (manual)
- Parsing for syntactic correctness (automated)
- Type-checking for semantic consistency (automated)
- Simulation/animation based on the specification; This is only possible if the language provides an execution option
- Theorem proving and proof-checking for logical anomalies (interactive)

1. Consistency within each Specification

The purely definitional style adopted and the strong typing mechanisms of PVS gave us strong assurance concerning the internal consistency of the specifications. If all the expressions that can be proven in our specification using the underlying logic are true then the logic is sound. The type checking conditions to be proved are generated to assure sound definitions and demonstrate semantic consistency. The type checking system offers some logical consistency from the type checker, however most of it is dependent on the manual inspection by the user. Completeness is when the specification identifies all contingencies and specifies appropriate behavior for all cases. Verification tools are used as an adjunct not a replacement for standard quality assurance methods. Formal methods are not a panacea, but can increase confidence in a system's reliability if applied with care and skill. PVS, a verification tool is very useful for consistency checks, but cannot assure completeness of a specification. Developing consistent and complete specifications is a complex, time consuming, and difficult process. In many applications, the effort required to make the specifications both consistent and complete is not warranted, and a consistent set of specifications is often sufficient [Hei96].

2. Consistency between the FTLS and the Formal Security Policy Model

Whether the FTLS was consistent with the Formal Security Policy Model remains unanswered since the proper inter-level mapping theorems that provide this consistency checking could not be constructed. If the appropriate mapping theorems between these two specifications could be developed, then one could verify that there were no inconsistencies between the abstract design of the implementation (FTLS) and the *State* specification. Also, since the system was unable to construct inter-level mapping from our specifications, we could not show a proof of correspondence between the *FTLSSpec* and *State*. Otherwise, bugs and unintended functionality could become known and then corrected early in the design phase of the system. Currently, the level of assurance is low since one can only verify the semantic and logical consistency within a specification. In addition, since PVS does not provide executable specifications, one cannot simulate the specifications to see whether the required properties hold or to really analyze whether there is logical consistency.

D. FUTURE WORK

1. Solving the Inter-level Mapping Problem

Future research should modify our given specifications so that they satisfy the ideal inter-level mapping construction as seen in Figure 9. Whether this problem is solvable remains questionable. Our empirical study should be applied to the next qualified verification system, ACL2, and comparisons should be made to determine the most suitable system for specific applications.

2. Background of A Computational Logic of Applicative Common Lisp (ACL2)

a. Introduction

ACL2 can be used as a programming language, a specification language, a modeling language, a formal mathematical logic, a semi-automatic theorem prover, and more. It basically is a mathematical logic together with a mechanical theorem prover to help one reason in the terms of the logic. ACL2 is programmed in and supports the logic of a large applicative ("side effect free") subset of Common Lisp. In other words, ACL2 is a very small subset of full Common Lisp. Roughly speaking, a language is applicative if it follows the rules of function application [Moo03].

b. Semantics

ACL2's underlying logic is quantifier free first order logic. The semantics of ACL2 include macros that can be extended with a single-threaded state, fast applicative arrays, and property lists. The macro facility of Common Lisp can be used to make specifications more succinct and easier to grasp. The formal models that are written in ACL2 are executable except when there are undefined functions. It is possible to introduce an undefined function whose value is constrained to be some object satisfying a certain formula, provided such an object exists. This then extends the ACL2 logic to full first order logic. In addition, it is possible to introduce a function whose "body" is a universally (or existentially) quantified formula [Moo03].

ACL2 supports five disjoint kinds of data objects[Kau02]:

- Numbers: 0, -123, 22/7, #c(2 3)
- Characters: #\A, #\a, #\$, #\Space
- Strings: "This is a string."
- Symbols: nil, x-pos, smith::x-pos
- Conses – (1 .2), (a b c), ((a . 1) (b . 2))

The primitive functions of ACL2 can be categorized as:

- Boolean (e.g. (and p1 p2…) : Logical conjunction operator
- Arithmetic (e.g. (+ x y) : Addition
- Characters (e.g. (char – code char) : Convert character to integer
- Strings (e.g. (length str) : Length of string (or list)
- Symbols (e.g. (symbol-name sym) : Name (string) of symbol

- Cons Pairs and Lists (e.g. (cons x y) : Construct an ordered pair

c. Guards/Other Features

ACL2 presents itself to the user as "read-eval print loop" where it repeatedly reads an expression from the user, evaluates it, and prints the result. The expressions typed into the read-eval print loop are called top level expressions and these expressions are not allowed to contain unbound variable symbols. ACL2 provides the ability for functions to compute and return multiple results and permits the use of declarations in certain expressions. These declarations are used to inform the Lisp compiler and the ACL2 system about pragmatic issues. In addition, the Definitional Principle allows the user to add axioms defining new function symbols, under conditions that insure that the soundness of the logic is preserved. ACL2 includes the use of packages, encapsulation, books, and theories that provide name and rule scoping. Since ACL2 is a programming language where hints and prover advice can be expressed and codified. ACL2 provides a powerful type-like mechanism called "guards" which can be used to assure that functions are "well typed". Guard verification provides mechanized support for proving that functions are used in compliance with Common Lisp.

d. Theorem Prover

The automatic theorem prover is driven by an incrementally constructed database of previously proved theorems. Decision procedures are for propositional calculus, equality, and linear arithmetic. There are integrated heuristics for congruence-based rewriting, backwards and forward chaining, destructor elimination, generalization, and induction. In addition, there are user supplied meta-theoretic simplifiers, an elaborate hint mechanism including the use of Binary Decision Diagrams (BDDs), and a built-in interactive "proof checker". The ACL2 prover is powerful and versatile with respect to problems couched in its "native" domain of recursively defined functions[You96*]. Since ACL2 is both an executable programming language and a specification language for the theorem prover, it allows the theorem prover to argue about its own code [Kro02].

e. Concerns

ACL2 does not support the hierarchical organization of theories, operator theories, and derived rules [Zha98]. It also is not based on higher order logic, and therefore may not be suitable for a wider range of applications. Since ACL2 is not only a programming language but also a logic, there are some restrictions on function definitions. For example, “global variables” are not allowed, any function used in the body other that the ones being defined must have been introduced earlier, and recursive definitions must be proved to terminate. While many programming languages have expressions, statements, blocks, procedures, or modules, etc., ACL2 just has expressions. The lisp syntax is sometimes a major stumbling block for many new users of ACL2, however it has the advantages of being completely unambiguous without any ancillary precedence rules [You96].

3. ACL2 vs. PVS

a. Expressiveness

i. Types

The primary goal of a type system is to ensure language safety by ruling out all untrapped errors in all program runs [Car97]. The declared goal of a type system is usually to ensure good behavior of all programs, by distinguishing between well typed and ill typed programs. Types and subtypes correspond to basic constructs of specification languages and programming languages. Support for types and subtypes is preferable and there are many different kinds of subtypes.

- Predicate subtype
- Dependent subtype
- Syntactic subtype
- Semantic subtype

Of all languages, PVS has the most expressive type system that consists of a base type and a predicate. The base type is a basic type such as the number type or an enumerative type, or combinations of basic types (tuples, records,…). Subtyping helps for writing understandable and concise specifications. ACL2 does not support a type system and its syntax is that of Common Lisp. Lisp derives much of its expressive utility from its weak typing, even though considerable execution efficiency can be gained from type declarations [You96*].

ii. Underlying Logic/Syntax

Higher order logic ensures that the specification language applies to the widest range of applications. The use of a more restrictive logic such as first order logic means that more effort sometimes must be invested in the specification but is often repaid with increased automation in the proof. In addition, higher order logic is elegant but seldom necessary, and higher orderness may be a notational convenience[You96*]. The elegance gained in specification must be weighed against the potential additional proof burden. Most lambda terms can be translated directly into recursive functions though often at the cost of additional parameters. In addition, sometimes the appropriate recursive function serves the purposes of a quantifier. ACL2 is essentially quantifier-free, first order logic. It is not as usable as PVS and has limited or no support for arbitrary quantification. PVS offers higher order functions, strong typing, lambda abstraction, and full quantification. These uses can easily be translated into simpler logical constructs that facilitate more automated proof discovery.

The syntax of ACL2 is that of Common Lisp, macros however make this syntax extremely malleable. ACL2 has both recursive and induction function in its syntax. ACL2 supports the macro facility of Common Lisp. Macros provide a powerful abbreviation facility and can be used to make specifications much more succinct and easier to grasp [You96*]. All the functions evaluate their arguments. ACL2 also supports the introduction of axiomatically constrained function symbols that have no executable counterpart. In ACL2, a macro package can be introduced to add enumerated types to a specification. Record structures can be easily represented in ACL2 using the available def-structure macros. Also, ACL2 integrates its untyped logic with guards to insure compliance with efficiently executable raw Lisp implementations. ACL2 provides execution of definitions and encourages concrete, efficient models. PVS cannot simulate machine execution conveniently but supports higher-order logic and encourages specifications unburdened by irrelevant detail.

b. Theorem Prover

The proof of a theorem is an unbroken chain of inferences that connect that theorem to axioms. Both ACL2 and PVS support the ability to do backward proofs, forward proofs, proof by rewrite, proof by induction, and proof by cases. In addition, ACL2 is able to perform proofs by contradiction. Theorem proving tools that are based on first order logics become very restrictive, therefore tools that are based on higher order logic such as PVS can be more desirable

ACL2 provides several automatic proof techniques that the user programs by proving theorems and then adding these theorems to the theorem prover database. The style of these proofs has an important benefit that is proof robustness. Since these proofs are "automatic", dramatic changes in the specification or system does not imply that the proofs will change drastically. Theorem provers that are based on first-order quantifier free logic are appropriate tools when working on large systems that have had corrected problems due to the mostly automatic approach to guiding the theorem prover.

PVS has support for automated reasoning, namely a simple rewriting system and a facility for constructing new proof commands. PVS can also be used to reason about computer systems in a robust style. Realistic proofs require robustness and PVS is capable of a proof style that fosters resilience in proofs about computer systems [Wil97].

E. CONCLUDING REMARKS

Our empirical study of PVS paved the way for future research of other verification systems. The methodology we used for the evaluation of PVS can be followed for the analysis of the next selected verification system. Since ACL2 was our second choice for our empirical study due to the relevant documentation, satisfied evaluation criteria, and our familiarity with the tool, its evaluation is the next step in future research. In the previous sections, ACL2's background and comparisons with PVS has been introduced. In this section, the proper methodology for an empirical study of ACL2 will be provided as well as a table summarizing the features of ACL2 that should be observed in the research process. The summation will be based on the features of PVS in comparison to ACL2. This is due to the fact that the main purpose of our work was to introduce a verification system that would be suitable for the project HANNAH (High Assurance Network Authenticator), which involves the creation of a high assurance trustworthy authenticator on a network. The policy we have used in our paper is a simplified version of a complex policy that may be used when implementing HANNAH. HANNAH requires the use of a verification system, and therefore relies on the results of the evaluation of the verification systems. Therefore, more verification systems need to be evaluated to figure out the most appropriate verification system that should be applied to HANNAH. By following the same methodology used in our study, a better analysis can be conducted to compare the different verification systems.

1. Methodology for Future Work

First, our formal security policy model should be used when constructing the formal security policy specification. When developing the specification in the language of ACL2 that is supposed to represent the formal security policy model and the FTLS, there are specific elements, functions, theorems, etc. that are required.

The formal security policy specification should have the following elements:

- Subjects
- Objects
- Access Attributes {read, write}
- Security Levels (Top Secret, Secret, Classified, Unclassified)

The formal security policy specification should have the following components:

- Current Access set
- Access Matrix
- Security Level function – Mapping Subjects and Objects to Security Labels

The formal security policy specification should have the following properties:

- A subject *s* is allowed to read an object *o only if* the security label of *s* dominates the security label of *o* - *n*o read up and applies to *all subjects*
- A subject *s* is allowed to write an object *o only if* the security label of *o* is equal to the security label of *s*
- Strong tranquility

The formal security policy specification should have the following rules:

- State transition operators
    - Add to and delete from the current set of accesses

The formal security policy specification should have the following theorems and proofs:

- The Basic Security Theorem that states that if the initial state (where state is the current set of accesses) is secure, and every state that results from some operation is secure, then every state is secure.

The formal top-level specification should have the following elements:

- ProcessID
- Memory Block
- Access Attributes {read, write}
- Security Levels (Top Secret, Secret, Classified, Unclassified)

The formal top-level specification should have the following components:

- Component that maps to the Current Access Set
- Component that maps to the Access Matrix
- Security Level function – Mapping Processes and Memory Blocks to Security Labels

The formal top-level specification should have the following properties:

- A process *p* is allowed to read an a memory block *m only if* the security label of *p* dominates the security label of *m* - *n*o read up and applies to *all processes*
- A process *p* is allowed to write a memory block *m only if* the security label of *p* is equal to the security label of *m*
- Strong tranquility

The formal top-level specification should have the following rules:

- State transition operators
    - Add or delete a memory block from a process

The formal top-level specification should have the following theorems and proofs:

- Theorems that prove the inter-level mapping between the FTLS and the formal security policy model specification

Learning the specification language of ACL2 and understanding how to use the theorem prover can take some time. However, the essential requirements of the specifications as well as our PVS specifications themselves are given as a reference that can reduce the amount of time and work. Choosing an appropriate verification system is essentially based on the preferences of the user of the system. It is the user whose criteria should be satisfied since he is the one interacting with the system. For example, a user with knowledge of Lisp has an advantage when using ACL2 since that person does not have to learn the specification language.

If the inter-level mapping problem cannot be resolved using the PVS system, then it should not be considered as an eligible system for the specified applications. However if both ACL2 and PVS support inter-level mapping then the criteria in Table 6 should be weighed to determine how the selection process should be handled. Once inter-level mapping is demonstrated, one can extrapolate from these results a conclusion about the usefulness of a verification system for the verification of real world systems. The work that has been done in this thesis demonstrates new research that has not been addressed in any PVS documentation that we have read. Therefore, the problems we have come upon as well as the ideas we have brought forth should be assessed and used in future work.

	ACL2	***PVS***
Age	*Current Version 2.7 in 2002 first developed around 1994*	*Current Version 3.1 in 2003, first developed around 1992*
Purpose	*General purpose theorem prover*	*General purpose theorem prover*
Implementation Language	*Untyped Common Lisp –* __?__	*Common Lisp – not very difficult to understand the theorem prover commands or messages*
Resource Requirements	*ACL2 works on the Unix, some variants including Linux, and Macintosh OS. It is built on top of any of the following Common Lisps: Allegro, GCL (Gnu Common Lisp) [or, AKCL], Lispworks, Lucid, and MCL (Macintosh Common Lisp)*	*PVS 3.1 is currently available only for Sparc machines with Solaris 2 and Intel x86 Machines with Linux compatible with Redhat 5 or later.*
User-Friendly	*Automatic theorem prover, user defined proof strategies, provides decision and inference strategies and previously proved theories, theorem prover can argue about its own code since ACL2 is both an executable programming language and a specification language* __?__	*Proofs are generated interactively, user defined proof strategies can also be invoked, interface is easy to understand, theorem prover command are straightforward and proof construction is not difficult, provides decision and inference strategies and previously proved theories, specification language and implementation language are not difficult to understand*
User Interface	*Emacs -* ___?___	*Gnu Emacs, Xemacs – Requires planning of proof strategy however organized in proof structure*

Table 6. Comparison between ACL2 and PVS (Questions Marks to be answered by future research)

	ACL2	***PVS***
User Presentation Language	*Common Lisp* __?__	*Its own presentation language – Able to express the necessary properties, succinct and easy to understand*
Consistency of Specifications	*Yes, Interactive consistency -* ___?___	*Yes, automated consistency checking of specifications – Type checking system*
Executable Specifications	*Yes, specifications may be executed in an underlying implementation of Common Lisp; thus can build executable specifications*	*No*
Multiple Levels of Abstractions	*Yes -* __?__	*Yes – using the IMPORTING clause to map one specification to another*
Expressiveness	*First Order Logic – results in unambiguous specifications and can increase the automation of the proof, the macros facility can be used to make specifications more succinct and easier to grasp; No Type System -* __?__	*Higher Order Logic - wider range of applications, more expressive, elegant but most of the time expressions in HOL can also be expressed in FOL; Type System - increases amount of flexibility and increases the precision of the specification System*

Continuation of Table 6 –Comparison between ACL2 and PVS

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A: AGE

ACL2

Current Version 2.7 2002, first started around 1994

AutoFOCUS

First developed in 1996

Coq

Current version 7.3.1 2002, first started 2001

Elf/Twelf

Current Version 1.3, first started in 1998

HOL

Latest Version now called HOL 4, First version came out in 1988

IMPS

Current version 2.0, first started in 1990

Isabelle

Current Version 2002, first distributed in 1986

Nuprl

Current Version 2002, first developed in 1984

Otter

Current Version 3.1 in 2000, first started in 1995

PVS

Current Version 3.1 in 2003, first developed in 1992

SpecWare

Phase 2

STeP

Developed in 1994

TAME

Developed around 1997

TPS

Current Version 2000, first developed in 1990

Vienna

Developed around 1978 – not sure if updated since 1996

Z/Eves

Current Version 2.1 in 2002, first started in 1990

APPENDIX B: PURPOSE

ACL2

ACL2 is a general purpose theorem prover that can be used to verify every step of a proof in almost any mathematical domain, from real analysis to circuit design. It is a stable and robust system designed to tackle industrial-size verification projects

AutoFOCUS

AutoFocus is based on formal methods of systems engineering. It will serve as start point and evaluation means for further tool concepts for the specification and development of distributed systems.

Coq

Coq is used for extracting programs from proofs.

Elf/Twelf

Elf/Twelf is a system based on predicative type theory that can be used to specify and reason about logics, languages, type systems, etc.

HOL

The HOL system has a wide variety of uses from formalizing pure mathematics to verification of industrial hardware. Academic and industrial sites world-wide are using HOL.

IMPS

IMPS is a interactive mathematical proof system intended to provide organizational and computational support for the traditional techniques of mathematical reasoning.

Isabelle

Isabelle allows for single-step proof construction and provides control structures for expressing search procedures. Isabelle also provides several generic tools, such as simplifiers and classical theorem provers, which can be applied to object logics.

Nuprl

Nuprl is a computer system which provides assistance with problem solving. It supports the interactive creation of proofs, formulas, and terms in a formal theory of mathematics.

With it one can express concepts associated with definitions, theorems, theories, books and libraries.

Otter

Otter is currently an automated deduction system. Otter is designed to prove theorems stated in first-order logic with equality.

PVS

PVS is a general purpose theorem proving tool.

SpecWare

Specware is a next-generation environment supporting the design, development and automated synthesis of scalable, correct-by-construction software.

STeP

STeP is a tool for the computer aided formal verification of reactive systems, including real-time and hybrid systems based on their temporal specification.

TAME

A major goal of TAME is to allow a software developer to use PVS to specify and prove properties of an I/O automaton efficiently and without first becoming a PVS expert.

TPS

TPS is an automated theorem-prover for first-order logic and type theory.

Vienna

Vienna is a collection of techniques for the formal specification and development of computing systems. It supports the top-down development of software systems specified in a notion suitable for formal verification.

Z/Eves

Z/Eves is a formal methods tool to incorporate a unique set of technologies. It is a proof obligation generator, an automated deduction system, and interactive theorem prover.

APPENDIX C: IMPLEMENTATION LANGUAGE

ACL2

Untyped Common Lisp

AutoFOCUS

Java

Coq

ML

Elf/Twelf

Standard ML

HOL

Standard ML

IMPS

Common lisp

Isabelle

Standard ML

Nuprl

ML

Otter

C

PVS

Common Lisp

SpecWare

PHASE 2

STeP

ML

TAME

Interface to PVS

TPS

Common Lisp

Vienna

VDM-SL, VDM++

Z/Eves

Common Lisp

APPENDIX D: RESOURCE REQUIREMENTS

ACL2

ACL2 works on Unix and some variants including Linux, and Macintosh OS

AutoFOCUS

Unix platforms

Coq

The current stable version of Coq is the 7.3.1. It currently is available for Unix (including Mac OS X) and Windows 95/98/NT systems. It also runs on several operating systems including Linux, Solaris, and Windows.

Elf/Twelf

Elf is implemented in Standard ML of New Jersey, which runs on a variety of architectures; it requires version 0.72 or greater.

HOL

Unix and Windows machines

IMPS

IMPS runs on Linux and Solaris platforms. IMPS 2.0 should work with most versions of Common Lisp.

Isabelle

Unix platforms; A minimal Isabelle installation requires only bash and perl (usually provided by the operating system), and a suitable implementation of Standard ML; A comfortable Isabelle working environment demands further user interface support.

Nuprl

Nuprl should run in any Common Lisp with CLX. There are also interfaces for Symbolics Lisp machines and Suns running the SunView window systems.

Otter

It has been used mostly on UNIX systems, but limited versions also run on PCs and Macintoshes.

PVS

PVS 3.0 is currently available only for Sparc machines with Solaris 2 and Intel x86 and for machines with Linux compatible with Redhat 5 or later.

SpecWare

Phase 2

STeP

STeP runs on SUN sparc 20, and UltraSparc under SOLARIS, DEC Alpha under IRIX, SGI under IRIX5, and x86 under LINUX

TAME

Same as PVS

TPS

Unix and Linux systems, and to some extent under Windows

Vienna

Unix platforms

Z/Eves

Z/Eves runs on Linux, Windows 95/98/NT and Solaris

APPENDIX E: USER-FRIENDLY

ACL2

It is not easy to get ACL2 to prove hard theorems. A user must understand the model, ACL2 as a mathematical logic, and be able to construct a proof (in interaction with ACL2). ACL2 will help construct the proof, but its primary role is to prevent logical mistakes. A user's responsibility is the creative burden or the mathematical insight why the model has the desired property.[Moo03]

AutoFOCUS

AutoFOCUS offers user support by using a specification pattern, a model-based editor, and consistency checks.

Coq

Coq allows for interactive construction of proofs.

Elf/Twelf

Elf/Twelf allows the user to program algorithms and express their correctness proof within the same language. It is not well suited for interactive development of theories.

HOL

Beginning users can rapidly start doing proofs using HOL. It is similar to PVS since it combines model checking with user guided interactive proof. HOL users have to learn some ML before using the system.

IMPS

IMPS provides user guided interactive proofs. The course of machine deduction is orchestrated and controlled by the user. Users can formulate mathematical concepts and arguments in a natural and direct manner. Theory interpretations between the theory multiples are created when needed by the user. The user has great freedom to decide in order in which he wants to work on different subgoals. It is an effective tool to a wide range of mathematically educated users.[Far95]

Isabelle

Isabelle allows for single step proof construction. Beginners can get by with a small repertoire of commands and a basic knowledge of how Isabelle works.

Nuprl

Nuprl supports interactive creation of proofs, formulas, and terms in a formal theory of mathematics. It has an interactive style of proof checking that characterizes Nuprl. In this system it is impossible to develop an incorrect proof. It has characteristics of an intelligent computer system in that it provides its users w/a facility for writing proof generating programs in ML.[Koh99]

Otter

Otter is an automated deduction system that is designed to prove theorems stated in first order logic with equality.

PVS

In PVS proofs are generated interactively and user defined proof strategies can also be invoked.

SpecWare

PHASE 2

STeP

In STeP, proofs can entirely be automatic or can be user directed.

TAME

TAME provides a template that the user completes to specify an I/O automaton and a set of proof steps natural for humans to use for proving properties of automata. Each proof step is implemented by a PVS strategy and possibly some auxiliary theories that support that strategy. Users can create proofs using "natural" or automatic proof steps, without learning the details of the PVS proof steps. TAME also provides better user feedback that that provided by PVS.[Arc01]

TPS

TPS can be proved automatically, interactively, or in a mixture of these modes. The automatic mode is quite primitive since it deals with equality.

Vienna

In Vienna, a small change between concrete specifications and code leads to a straightforward implementation, automatic code generation, and a descriptive language. However, VDM is still relatively difficult to use.

Z/Eves

NEVER is the automated deduction component of EVES. It is an interactive theorem prover that is capable of automatically performing large proof steps, yet can be finely directed by the user. NEVER is neither a fully automatic nor an entirely manual theorem prover. Although NEVER provides powerful deductive techniques for the automatic proof of theorems, it also includes simple user steps that permits its use as a system more akin to a proof checker than a theorem prover. The possible fine control of the prover allows users to closely investigate proof strategies and determine why, for example, proofs are failing.[Koh99]

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX F: USER INTERFACE

ACL2

Emacs

AutoFOCUS

Project browser for the organization of the specification documents of the single project, and one editor for each specification document

Coq

Standard teletype-like shell window; does not use any special interface such as Emacs or Centaur

Elf/Twelf

Standard ML interface or Emacs mode

HOL

Xhol, chol, emacs

IMPS

Emacs-based interface

Isabelle

ML/user interface

Nuprl

Xwindows

Otter

Xwindows

PVS

Gnu Emacs, Xemacs

SpecWare

Phase 2

STeP

Xwindows

TAME

Friendly user interface - interface to PVS

TPS

Phase 2

Vienna

Phase 2

Z/Eves

Graphical interface-allows two specifications to be entered and edited

APPENDIX G: USER PRESENTATION LANGUAGE

ACL2

Common Lisp

AutoFOCUS

Phase 2

Coq

Phase 2

Elf/Twelf

Phase 2

HOL

Phase 2

IMPS

Phase 2

Isabelle

Phase 2

Nuprl

Common Lisp

Otter

Phase 2

PVS

It has its own presentation language

SpecWare

Phase 2

STeP

Phase 2

TAME

Phase 2

TPS

Phase 2

Vienna

Phase 2

Z/Eves

Phase 2

APPENDIX H: CONSISTENCY OF SPECIFICATIONS

ACL2

Yes, interactive consistency

AutoFOCUS

Phase 2

Coq

May cause ambiguity

Elf/Twelf

Consistency is unclear

HOL

Yes

IMPS

Phase 2

Isabelle

Yes

Nuprl

Yes, has consistency proof of formal specifications

Otter

Phase 2

PVS

Yes, automated consistency checking of specifications

SpecWare

Phase 2

STeP

Phase 2

TAME

Yes, automated consistency of specifications

TPS

Phase 2

Vienna

Phase 2

Z/Eves

Yes, has a consistency checker

APPENDIX I: EXECUTABLE SPECIFICATIONS

ACL2

Yes, specifications may be executed in an underlying implementation of Common Lisp; thus can build executable specifications

AutoFOCUS

Phase 2

Coq

Yes, there are papers about efficient reasoning about executable specifications

Elf/Twelf

Many specifications are not executable under the traditional logic programming semantics and performance may be hampered by redundant computation.

HOL

Yes, hol, hol.bare, hol.unquote, and hol.bare.unquote

IMPS

Phase 2

Isabelle

Phase 2

Nuprl

Yes, functional language for executable specifications

Otter

Phase 2

PVS

No

SpecWare

Phase 2

STeP

Phase 2

TAME

Phase 2

TPS

Phase 2

Vienna

You can write executable and non-executable specifications in VDM-SL

Z/Eves

Phase 2

APPENDIX J: MULTIPLE LEVELS OF ABSTRACTION

ACL2

Yes

AutoFOCUS

Phase 2

Coq

Phase 2

Elf/Twelf

Phase 2

HOL

Yes

IMPS

Yes

Isabelle

Phase 2

Nuprl

Phase 2

Otter

Phase 2

PVS

Yes

SpecWare

Phase 2

STeP

Phase 2

TAME

Phase 2

TPS

Phase 2

Vienna

Phase 2

Z/Eves

Phase 2

APPENDIX K: EXPRESSIVENESS

ACL2

ACL2 blends arithmetic decision procedures with rewriting techniques and it supports rational numbers, complex rationals, character objects, character strings, and symbol packages. Formal models written in ACL2 are usually executable unless they are undefined functions. ACL2 has a powerful type – like mechanism called "guards" which can be used to assure that functions are "well-typed". Since it is programmed in the same logic it supports, it can ensure that the logic is a practical means of building large formal systems. Its logic is first-order quantifier free, its semiautomatic, and it uses lemmas as guidance. In addition, it has many design procedures (prepositional calculus, equality, arithmetic) and heuristics. Also, it has a built-in interactive "proof checker".[Koh99]

AutoFOCUS

AutoFOCUS has formal, logical foundations, mathematical models and methods for distributed systems. It is based on traces and stream processing functions(combine functions and states). It creates consistency checks and supports simple temporal logic.[Tum03]

Coq

Coq is a higher-order proof system where proofs and terms are in a pure functional language. The basic specification language is called Gallina, in which formal axiomatisations may be developed. It is a non-conservative extension of the Calculus of Constructions with inductive types and is well adapted to inductive reasoning and has a powerful type system. It can also be described as a directed tactics theorem-prover with a set of predefined tactics, including an auto tactic that tries to apply precious lemmas declared as hints. Coq offers program extraction where the logic mixes a constructive logic and a classical logic. The system automatically extracts the constructive contents of proofs as an executable ML program.[Koh99]

Elf/Twelf

Elf/Twelf is a LF logical framework based on a predicative type theory. It lends itself to the very direct specification of programming languages, type systems, logics, etc. One then can program algorithms and express their correctness proof within the same language. Features that make it unique are the internal notion of deduction and the expressive language of types (which include dependent types). These together can be exploited to implement the meta-theory of programming languages, compiler, logics, etc. All data is developed and stored as programs. It employs a sophisticated term reconstruction algorithm that allows much of the input information to be elided without any syntactic restrictions.[Koh99] [Pfe03]

HOL

HOL's most outstanding feature is its high degree of programmability through the *meta-language* ML. The scope of type variables is the current term in HOL and type-checking includes simple typechecking. In addition, it has more emphasis on logical foundations and less on usability. HOL does not support typechecking with respect to predicate membership. A primary use of HOL is the building of special purpose proof infrastructure. Users can add external tools to HOL and embed languages just by programming in ML. The HOL System is an environment for interactive theorem proving in a higher order logic.[Koh99] [Iyo03]

IMPS

The logic of IMPS is based on a version of simple type theory with partial functions and subtypes. Automated analysis, computing with theorems, higher order logic, theory interpretation, mathematical specification and inference are performed relative to axiomatic theories. IMPS provides relatively large primitive inference steps to facilitate human control of the deductive process and human comprehension of the resulting proofs. There is a library containing almost a thousand repeatable proofs that covers significant portions of logic, algebra and analysis. [Far95]

Isabelle

Isabelle is a generic theorem prover instantiated to support reasoning in several object logics: 1. First order logic, constructive, and classical versions, 2. Higher order logic compared to HOL, 3. Zermelo Fraenkel set theory, 4. Extensional version of Martin Lofs Type Theory, 5. The classical first order sequent calculus, 6. The modal logics T, S4, S43, 6. The logic for computable functions; Isabelle borrows ideas of LCF where formulas are ML values, theorems belong to abstract types, and tactics support backward proofs. However, object level rules are represented by terms not functions. Isabelle theorem proving can involve writing ML code. [Pau02]

Nuprl

The logic has a constructive semantics in that the meaning of propositions is given by rules of use and in terms of computation. One of the salient features of it is that the logic and the system take account of the computational meaning of assertions and proofs. For instance, given a constructive existence proof, the system can use the computational information in the proof to build a representation of the object which demonstrates the truth of the assertion. It supports an interactive environment for text editing, proof generation and function evaluation. It has an interactive style of proof checking that characterizes it. In this system it is impossible to develop an incorrect proof. It has characteristics of an intelligent computer system in that it provides its users with a facility for writing proof--generating programs in a metalanguage, ML. The style of the logic is based on the stepwise refinement paradigm for problem solving in that the system encourages the user to work backwards from goals to subgoals until one reaches what is known.[Koh99][Cor03]

Otter

Otter supports first-order logic with equality. Otter's inference rules are based on resolution and paramodulation, and it includes facilities for term rewriting, term orderings, Knuth-Bendix completion, weighting, and strategies for directing and restricting searches for proofs. Otter can also be used as a symbolic calculator and has an embedded equational programming system. The main application of Otter is research in abstract algebra and formal logic. Otter and its predecessors have been used to answer many open questions in the areas of finite semigroups, ternary Boolean algebra, logic calculi, combinatory logic, group theory, lattice theory, and algebraic geometry.[Arg03]

PVS

PVS supports classical higher-order logic with functions, sets, records, tuples, predicate subtypes, dependent typing, and theories with type and individual parameters. Axioms may be introduced freely and definitions (which may be recursive and include recursively-defined abstract data types and inductively defined predicates) provide conservative extension. It is an expressive specification language that allows concise and natural specifications across a wide range of problem domains. It also has a rich type system and very strict typechecking allows much of the specification to be embedded in the types with proof obligations generated automatically by the typechecker. There is a synergistic interaction between theorem proving and typechecking. PVS is an effective interactive theorem prover that combines powerful arithmetic decision procedures and other "large" primitive inference steps. PVS has the ability to display "humanly readable" proofs.[Koh99][Sri03]

SpecWare

SpecWare supports automation of component-based specification of programs using a graphical interface, incremental refinement of specifications into correct code in various target languages (e.g. C++, LISP, Ada, Cobol), design and synthesis of software architectures/frameworks, and design and synthesis of algorithm schemas. In addition, it supports design and synthesis of reactive systems, data-type refinement, program optimization, recording and experimenting with different design decisions, domain-knowledge capture, verification and manipulation.[Kes03]

STeP

STeP integrates methods for deductive and algorithmic verification, including model checking, theorem proving, automatic invariant generation, abstraction and modular reasoning.[Sta03]

TAME

TAME provides two types of strategies: strategies for "automatic" proof and strategies designed to implement "natural" proof steps, i.e. proof steps that mimic the high-level steps in typical natural language proofs. It is a general-purpose higher-order logic theorem prover that uses standard logic.[Arc01]

TPS

TPS supports first and higher -order logic. It has facilities for searching for expansion proofs, translating these into natural deduction proofs, constructing natural deduction proofs, translating natural deduction proofs which do not contain cuts into expansion proofs, and solving unification problems in higher-order logic. It has a formula editor which facilitates constructing new formulas from others already known to TPS, and a library facility for saving formulas, definitions, and modes (groups of flag settings). It can operate in automatic, semi-automatic or interactive mode and it's logical language is that of typed lambda-calculus. It has two proving components: search for an expansion proof, and meta language for defining tactics and constructing natural deduction proofs. It is controlled by setting of over 150 user settable flags and it has a formula editor, library facility for saving formulas, definitions, and modes. [And00]

Vienna

VDM-SL has a formally defined semantics. The logic underlying this semantics is based on the Logic of Partial Functions (LPF). The definition of the semantics is given in a denotational style in the VDM-SL Standard. It has rules for data and operation refinement which allows one to establish links between abstract requirements specifications and detailed design specifications down to the level of code. It has a proof theory in which rigorous arguments can be conducted about the properties of specified systems and the correctness of design decisions. In addition, it uses a specification notation that is similar to Z. Its formal specifications is IS0-VDM – SL. [Ibm03]

Z/Eves

Z/Eves includes a graphical user interface that allows Z specifications to be entered, edited, and analyzed in their typeset form. It supports the incremental analysis of specifications and manages the synchronization of the analysis with modifications to the specification. In addition, it supports untyped first -order logic, without the conventional distinction between terms. EVES has a mechanism for defining new functions and every declaration must be proved to define a conservative extension of the theory in which it appears. There is also a library system that allows theories to be combined. It combines the power of its automatic capabilities (embodied in the reduction commands) with the ability of the user to have fine-grained control (using command modifiers and low level inference commands). We view the unique attributes of EVES as being the combination of (i) an expressive formal specification and programming language, (ii) practical automated deduction support, (iii) mathematical soundness, (iv) rigorous tool development, and (v) availability on workstations and PCs. [Koh99] [Ora03]

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX L: DEVELOPERS/SUPPORT

ACL2

Editors Kaufmann and Moore,
Contact persons:moore@cs.utexas.edu,kaufmann@aus.edsr.eds.com

AutoFOCUS

Department of Computer Science, TU Munchen

Coq

INRIA Rocquencourt and ENS Lyon

Elf/Twelf

Mark Pfenning at Carnegie - Mellon University
Contact: Frank Pfenning, School of Computer Science, CMU – there is a mailing list with announcement of new papers and implementation releases of Elf

IMPS

Mitre corporation

HOL

Cambridge university, computer laboratory

Isabelle

Cambridge University and TU Munich

Nuprl

Department or Computer Science, Cornell University

Otter

Argonne national laboratory

PVS

SRI International

SpecWare

Kestrel institute

STeP

Stanford University

TAME

Naval Research Laboratory

TPS

Carnegie Mellon University

Vienna

IBM Vienna Laboratory

Z/Eves

ORA Canada

APPENDIX M: *SEC_THEORY*, *SLABELS*, *STATE* SPECIFICATIONS

```
Sec_theory[Access_State: TYPE, TransformInstance: TYPE, seq:
sequence[Access_State], SecState:[Access_State -> bool], Transform:
[TransformInstance, Access_State -> Access_State]]: THEORY
 BEGIN
  ASSUMING
   st: VAR Access_State

   x: VAR TransformInstance

   n: VAR nat


% Assumptions needed to form an inductive proof
   transition_state_secure: ASSUMPTION SecState(st) => SecState(Transform(x, st))

   seq_0_secure: ASSUMPTION SecState(nth(seq, 0))

% there exists a transform that will take the system from state n to n+1
   seq_transform: ASSUMPTION
     EXISTS x: nth(seq, n + 1) = Transform(x, nth(seq, n))

  ENDASSUMING

% Every state in the sequence is a secure state
  seq_is_secure: THEOREM every(SecState)(seq)

 END Sec_theory

-----------------------------------------------------------------------------------------------------
SLabels: THEORY
  BEGIN

    Label: TYPE+ = {i:nat | i <=4 AND i > 0} CONTAINING 1

    U: Label = 1
    C: Label = 2
    S: Label = 3
  TS: Label = 4

  END SLabels

---------------------------------------------------------------------------------------------------
State[Subject: TYPE+, Object: TYPE+, (IMPORTING SLabels) slSubject:
    [Subject -> Label], slObject: [Object -> Label], Action: TYPE+,
    addit: Action, del: Action, no_op: setof[Action], Mode: TYPE+, rd: Mode, wr:
Mode ]: THEORY
 BEGIN

%  Assumptions needed to prove proof obligations
  ASSUMING
   AddDelNotNo_Op: ASSUMPTION
     NOT member(addit, no_op) AND NOT member(del, no_op)

   AddnotDel: ASSUMPTION NOT addit = del

   RdnotWr: ASSUMPTION NOT rd = wr

  ENDASSUMING

  Access: TYPE =
  [# u: Subject, f: Object, m: {x: Mode | x = rd OR x = wr} #]

  Access_State: TYPE = setof[Access]

  TransformInstance: TYPE = [# ac: Action, a: Access #]

 % Verifies that an access is secure for a specific mode by comparing security
 % labels of the subject and object
  SecureAccess(a: Access): bool =
     COND a`m = rd -> slSubject(a`u) >= slObject(a`f),
         a`m = wr -> slSubject(a`u) = slObject(a`f)
     ENDCOND

%  If all accesses in an Access_State are secure, then the Access_State is
% secure
  SecState(st: Access_State): bool =
  FORALL (a: Access): member(a, st) => SecureAccess(a)

% If a tranform is to add an access to an Access_State and the access to be added is
% secure then add that access to the Access_State.  If a transform is to delete an access
% from an Access_State and the access is already in the state, then remove the access
%  from the Access_State
  Transform(e: TransformInstance, st: Access_State): Access_State =
     COND e`ac = addit AND SecureAccess(e`a) -> add(e`a, st),
         e`ac = del AND st(e`a) -> remove(e`a, st),
         member(e`ac, no_op) -> st,
         ELSE -> st
     ENDCOND
 fs: sequence[Access_State]
 fs_0_secure: AXIOM SecState(nth(fs, 0))

 n: VAR nat

 fs_transform: AXIOM
   EXISTS (e: TransformInstance): nth(fs, n + 1) = Transform(e, nth(fs, n))

 transition_state_secure: AXIOM
   (FORALL (st: Access_State, e: TransformInstance):
     SecState(st) => SecState(Transform(e, st)))

 IMPORTING Sec_theory[Access_State, TransformInstance, fs, SecState,
                 Transform]


END State
----------------------------------------------------------------------------------------------------
```

State TCCs

```
  % Disjointness TCC generated (at line 47, column 6) for
   % COND a`m = rd -> slSubject(a`u) >= slObject(a`f),
   %      a`m = wr -> slSubject(a`u) = slObject(a`f)
   % ENDCOND
 % proved - complete
SecureAccess_TCC1: OBLIGATION FORALL (a: Access): NOT (a`m = rd AND
a`m = wr);

% Coverage TCC generated (at line 47, column 6) for
   % COND a`m = rd -> slSubject(a`u) >= slObject(a`f),
   %      a`m = wr -> slSubject(a`u) = slObject(a`f)
   % ENDCOND
 % proved - complete
SecureAccess_TCC2: OBLIGATION FORALL (a: Access): a`m = rd OR a`m = wr;

% The disjointness TCC (at line 47, column 6) in decl SecureAccess for
   %  COND a`m = rd -> slSubject(a`u) >= slObject(a`f),
   %      a`m = wr -> slSubject(a`u) = slObject(a`f)
   % ENDCOND
 % was not generated because it simplifies to TRUE.

% The coverage TCC (at line 47, column 6) in decl SecureAccess for
   %  COND a`m = rd -> slSubject(a`u) >= slObject(a`f),
   %      a`m = wr -> slSubject(a`u) = slObject(a`f)
   % ENDCOND
 % was not generated because it simplifies to TRUE.
% Disjointness TCC generated (at line 57, column 6) for
   % COND e`ac = addit AND SecureAccess(e`a) -> add(e`a, st),
   %      e`ac = del AND st(e`a) -> remove(e`a, st),
   %      member(e`ac, no_op) -> st,
   %      ELSE -> st
   % ENDCOND
  % proved - complete
Transform_TCC1: OBLIGATION
  FORALL (e: TransformInstance, st: Access_State):
    NOT ((e`ac = addit AND SecureAccess(e`a)) AND e`ac = del AND st(e`a)) AND
     NOT ((e`ac = addit AND SecureAccess(e`a)) AND member[Action](e`ac, no_op))
      AND NOT ((e`ac = del AND st(e`a)) AND member[Action](e`ac, no_op));

% The disjointness TCC (at line 57, column 6) in decl Transform for
   %  COND e`ac = addit AND SecureAccess(e`a) -> add(e`a, st),
   %      e`ac = del AND st(e`a) -> remove(e`a, st),
   %      member(e`ac, no_op) -> st,
   %      ELSE -> st
   % ENDCOND
  % was not generated because it simplifies to TRUE.

% Assuming TCC generated (at line 76, column 12) for
   % Sec_theory[Access_State, TransformInstance, fs, SecState, Transform]
   % generated from assumption Sec_theory.transition_state_secure
  % proved - complete
IMP_Sec_theory_TCC1: OBLIGATION
  FORALL (st: Access_State, x: TransformInstance):
    SecState(st) => SecState(Transform(x, st));

% Assuming TCC generated (at line 76, column 12) for
   % Sec_theory[Access_State, TransformInstance, fs, SecState, Transform]
   % generated from assumption Sec_theory.seq_0_secure
  % proved - complete
IMP_Sec_theory_TCC2: OBLIGATION SecState(nth[Access_State](fs, 0));

% Assuming TCC generated (at line 76, column 12) for
   % Sec_theory[Access_State, TransformInstance, fs, SecState, Transform]
   % generated from assumption Sec_theory.seq_transform
  % proved - complete
IMP_Sec_theory_TCC3: OBLIGATION
  FORALL (n: nat):
    EXISTS (x: TransformInstance):
      nth[Access_State](fs, n + 1) = Transform(x, nth[Access_State](fs, n));
--------------------------------------------------------------------------------------------------
```

Proofs for *Sec_theory* and *State* Specifications

Proof scripts for importchain of theory State:

Sec_theory.seq_is_secure: proved - complete [shostak](0.50 s)

```
(""
 (expand "every")
 (induct "n")
 (("1" (use "seq_0_secure"))
  ("2"
   (skolem!)
   (use "seq_transform")
   (skolem!)
   (use "transition_state_secure")
   (replace -2 (-1) rl)
   (propax))))
```

State.SecureAccess_TCC1: proved - complete [shostak](3.68 s)

```
("" (skolem!) (use "RdnotWr") (flatten) (simplify) (grind))
```

State.SecureAccess_TCC2: proved - complete [shostak](1.24 s)

```
("" (skolem!) (typepred "a!1`m") (propax))
```

State.Transform_TCC1: proved - complete [shostak](0.60 s)

```
("" (skolem!) (use "AddDelNotNo_Op") (use "AddnotDel") (grind))
```

State.IMP_Sec_theory_TCC1: proved - complete [shostak](2.46 s)

```
("" (skolem!) (use "transition_state_secure"))
```

State.IMP_Sec_theory_TCC2: proved - complete [shostak](0.64 s)

```
("" (use "fs_0_secure"))
```

```
State.IMP_Sec_theory_TCC3: proved - complete [shostak](2.90 s)

("" (skolem!) (use "fs_transform"))
```

APPENDIX N: *FTLSSPEC* SPECIFICATION

```
FTLSSpec[Action: TYPE+, addit: Action, del: Action, no_ops: setof[Action],
        Mode: TYPE+, rd: Mode, wr: Mode]: THEORY
 BEGIN

  IMPORTING SLabels

 % Axioms are used instead of assumptions since these principles should be satisfied
 % in the implementation
  AddDelNotNo_Ops: AXIOM
    NOT member(addit, no_ops) AND NOT member(del, no_ops)

  AdditNotDel: AXIOM NOT addit = del

% Read and write are the modes
  ReadNotWr: AXIOM NOT rd = wr

  MAXMEM: nat

  Memory: TYPE+ = {mem: nat | mem <= MAXMEM} CONTAINING MAXMEM

  MemoryExists: AXIOM FORALL (m: Memory): m > 0

% A memory block is represented by lower and upper bound values of memory
  MemBlock: TYPE+ = [# lb: Memory, ub: {x: Memory | x >= lb} #] CONTAINING (#
lb  := MAXMEM,  ub := MAXMEM #)

  SetofMB: TYPE+ = setof[MemBlock] CONTAINING {x: MemBlock    | x =
                (# lb := MAXMEM, ub := MAXMEM #)}

  slMemBlock: [MemBlock -> Label]

% Verifies that memory block do not overlap
  ValidMemBlocks(smb: SetofMB): bool =
     FORALL (e, b: MemBlock):
       NOT e = b AND member(e, smb) AND member(b, smb) =>
        (e`lb > b`ub) OR (b`lb > e`ub)

 % System memory blocks are memory blocks that are valid
  SystemMemBlocks: TYPE+ = {sysmb: SetofMB | ValidMemBlocks(sysmb)}
  CONTAINING {x: MemBlock | x = (# lb := MAXMEM, ub := MAXMEM #)}
% System memory is a fixed set of valid memory blocks
  SysMem: SystemMemBlocks
  ProcessID: TYPE+ FROM nat

% Each memory block instance has an associated mode
  MemBlockInstance: TYPE =
  [# mb: {x: MemBlock | member(x, SysMem)}, m: Mode #]

  ProcessMB: TYPE = setof[MemBlockInstance]

% Each process is associated with a ProcessID number and a set of memory blocks
% with their associated modes
  Processes: TYPE = [ProcessID -> ProcessMB]

  slPr: [ProcessID -> Label]

  Request: TYPE = [# ac: Action, p: ProcessID, mbi: MemBlockInstance #]

% Provides some inter-level mapping between the State specification and the FTLSSpec
%  specification
  IMPORTING State[ProcessID, MemBlock, slPr, slMemBlock, Action, addit,

               del, no_ops, Mode, rd, wr]

% Maps a component of FTLSSpec to TransformInstance, a component of State
% A TransformInstance's action corresponds to a request's action, a subject corresponds
%  to  a request's processed, an object corresponds with a request's memory block and a
% mode corresponds with a request's mode
  Proc_Trans_Inst(r: Request): TransformInstance =
     (# ac := r`ac, a := (# u := r`p, f := r`mbi`mb, m := r`mbi`m #) #)

  p: VAR Processes

% Maps a component of FTLSSpec to Access_State, a component of the State
%  specification
% Process_State maps processes that map to an Access_State where for every access in
% the Access_State there exists a memory block and ProcessID in which the access'
% subject corresponds with a ProcessID and the access' object corresponds with a
% memory block and the access' mode corresponds with the memory block's mode and
% the memory block is a member of that process associated with the ProcessID
  Process_State(p): Access_State =
     {a: Access | EXISTS (mbi: MemBlockInstance, PID: ProcessID):
         a`u = PID AND a`f = mbi`mb AND a`m = mbi`m AND member(mbi, p(PID))}

% Verifies that a request is secure by checking the access privileges a memory block
% and a process
  SecureRequest?(r: Request): bool =
     COND r`ac = addit ->
          ((r`mbi`m = rd AND slPr(r`p) >= slMemBlock(r`mbi`mb)) OR
           (r`mbi`m = wr AND slPr(r`p) = slMemBlock(r`mbi`mb))),
         ELSE -> TRUE
     ENDCOND

% A memory block can be added to a process if it is part of a secure request.
% A memory can be deleted from a process by just removing it from the process
  ProcTransform(r: Request, Pr: Processes): Processes =
     COND ac(r) = addit AND SecureRequest?(r) ->
          LAMBDA (x: ProcessID):
            IF NOT x = r`p THEN Pr(x) ELSE add(r`mbi, Pr(x)) ENDIF,
         r`ac = del ->
          LAMBDA (x: ProcessID):
            IF NOT x = r`p THEN Pr(x) ELSE remove(r`mbi, Pr(x)) ENDIF,
         ELSE -> Pr
     ENDCOND

% Verifies that all memory blocks of a process satisfy the security properties then it is a
% secure process
  SecProcess(p: Processes, pid: ProcessID): bool =
     FORALL (mbi: MemBlockInstance | member(mbi, p(pid))):
       COND mbi`m = rd -> slPr(pid) >= slMemBlock(mbi`mb),
           mbi`m = wr -> slPr(pid) = slMemBlock(mbi`mb)
       ENDCOND

% Verifies that if all processes are secure in the current state, then the state is secure
  SecProcesses(pr: Processes): bool =
     FORALL (pid: ProcessID): SecProcess(pr, pid)

SecProcesses(pr) => SecProcesses(ProcTransform(r, pr)))

END FTLSSpec
------------------------------------------------------------------------------------------------------
```

FTLSSpec Specification TCCs

```
% Subtype TCC generated (at line 16, column 56) for  MAXMEM
   % expected type  Memory
  % proved - complete
Memory_TCC1: OBLIGATION MAXMEM <= MAXMEM;

% Subtype TCC generated (at line 40, column 13) for
   % {x: MemBlock | x = (# lb := MAXMEM, ub := MAXMEM #)}
   % expected type  SystemMemBlocks
  % proved - complete
SystemMemBlocks_TCC1: OBLIGATION
```

```
 ValidMemBlocks({x: MemBlock | x = (# lb := MAXMEM, ub := MAXMEM #)});

% Assuming TCC generated (at line 57, column 12) for
   % State[ProcessID, MemBlock, slPr, slMemBlock, Action, addit, del,
   %       no_ops, Mode, rd, wr]
   % generated from assumption State.AddDelNotNo_Op
 % proved - complete
IMP_State_TCC1: OBLIGATION
 NOT member[Action](addit, no_ops) AND NOT member[Action](del, no_ops);

% Assuming TCC generated (at line 57, column 12) for
   % State[ProcessID, MemBlock, slPr, slMemBlock, Action, addit, del,
   %       no_ops, Mode, rd, wr]
   % generated from assumption State.AddnotDel
 % proved - complete
IMP_State_TCC2: OBLIGATION NOT addit = del;

% Assuming TCC generated (at line 57, column 12) for
   % State[ProcessID, MemBlock, slPr, slMemBlock, Action, addit, del,
   %       no_ops, Mode, rd, wr]
   % generated from assumption State.RdnotWr
 % proved - complete
IMP_State_TCC3: OBLIGATION NOT rd = wr;

% Disjointness TCC generated (at line 79, column 6) for
   % COND r`ac = addit AND SecureRequest?(r) ->
   %       LAMBDA (x: ProcessID):
   %         IF NOT x = r`p THEN Pr(x) ELSE add(r`mbi, Pr(x)) ENDIF,
   %     r`ac = del ->
   %       LAMBDA (x: ProcessID):
   %         IF NOT x = r`p THEN Pr(x) ELSE remove(r`mbi, Pr(x)) ENDIF,
   %     ELSE -> Pr
   % ENDCOND
 % proved - incomplete
ProcTransform_TCC1: OBLIGATION
 FORALL (r: Request):
   NOT ((r`ac = addit AND SecureRequest?(r)) AND r`ac = del);

% The disjointness TCC (at line 79, column 6) in decl ProcTransform for
   %  COND r`ac = addit AND SecureRequest?(r) ->
   %       LAMBDA (x: ProcessID):
   %         IF NOT x = r`p THEN Pr(x) ELSE add(r`mbi, Pr(x)) ENDIF,
   %     r`ac = del ->
   %       LAMBDA (x: ProcessID):
   %         IF NOT x = r`p THEN Pr(x) ELSE remove(r`mbi, Pr(x)) ENDIF,
   %     ELSE -> Pr
   % ENDCOND
  % was not generated because it simplifies to TRUE.

% Disjointness TCC generated (at line 92, column 8) for
   % COND mbi`m = rd -> slPr(pid) >= slMemBlock(mbi`mb),
   %      mbi`m = wr -> slPr(pid) = slMemBlock(mbi`mb)
   % ENDCOND
  % proved - incomplete
SecProcess_TCC1: OBLIGATION
  FORALL (p: Processes, pid: ProcessID,
        mbi: MemBlockInstance | member[MemBlockInstance](mbi, p(pid))):
   NOT (mbi`m = rd AND mbi`m = wr);

% The disjointness TCC (at line 92, column 8) in decl SecProcess for
   %  COND mbi`m = rd -> slPr(pid) >= slMemBlock(mbi`mb),
   %      mbi`m = wr -> slPr(pid) = slMemBlock(mbi`mb)
   % ENDCOND
  % was not generated because it simplifies to TRUE.

% The coverage TCC (at line 92, column 8) in decl SecProcess for
   %  COND mbi`m = rd -> slPr(pid) >= slMemBlock(mbi`mb),
   %      mbi`m = wr -> slPr(pid) = slMemBlock(mbi`mb)
   % ENDCOND
  % was not generated because it simplifies to TRUE.


--------------------------------------------------------------------------------------------------
```

Proof Commands for *FTLSSpec* Specification

Proof scripts for importchain of theory FTLSSpec:

```
Sec_theory.seq_is_secure: unchecked [shostak](5.29 s)

(""
 (expand "every")
 (induct "n")
 (("1" (use "seq_0_secure"))
  ("2"
   (skolem!)
   (use "seq_transform")
   (skolem!)
   (use "transition_state_secure")
   (replace -2 (-1) rl)
  (propax))))

State.SecureAccess_TCC1: proved - complete [shostak](3.68 s)

("" (skolem!) (use "RdnotWr") (flatten) (simplify) (grind))

State.SecureAccess_TCC2: proved - complete [shostak](1.24 s)

("" (skolem!) (typepred "a!1`m") (propax))

State.Transform_TCC1: proved - complete [shostak](0.60 s)

("" (skolem!) (use "AddDelNotNo_Op") (use "AddnotDel") (grind))

FTLSSpec.Memory_TCC1: proved - complete [shostak](0.03 s)

("" (tcc))

FTLSSpec.SystemMemBlocks_TCC1: proved - complete [shostak](0.14 s)

("" (tcc))

FTLSSpec.IMP_State_TCC1: proved - complete [shostak](0.03 s)

("" (use "AddDelNotNo_Ops"))

FTLSSpec.IMP_State_TCC2: proved - complete [shostak](0.07 s)

("" (tcc) (use "AdditNotDel"))

FTLSSpec.IMP_State_TCC3: proved - complete [shostak](0.72 s)

("" (use "ReadNotWr"))

FTLSSpec.ProcTransform_TCC1: proved - complete [shostak](0.15 s)

("" (skolem!) (flatten) (use "AdditNotDel") (grind))
```

```
FTLSSpec.SecProcess_TCC1: proved - complete [shostak](0.09 s)

("" (skolem!) (use "ReadNotWr") (grind))
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX O: PROOF OF THEOREM *SEQ_IS_SECURE* OF *SEC_THEORY* SPECIFICATION

```
seq_is_secure :

  |-------
{1}   every(SecState)(seq)

Rerunning step: (expand "every")
Expanding the definition of every,
this simplifies to:
seq_is_secure :

  |-------
{1}   FORALL n: SecState(nth(seq, n))

Rerunning step: (induct "n")
Inducting on n on formula 1,
this yields  2 subgoals:
seq_is_secure.1 :

  |-------
{1}   SecState(nth(seq, 0))

Rerunning step: (use "seq_0_secure")
Using lemma seq_0_secure,

This completes the proof of seq_is_secure.1.

seq_is_secure.2 :

  |-------
{1}   FORALL j: SecState(nth(seq, j)) IMPLIES SecState(nth(seq, j + 1))

Rerunning step: (skolem!)
Skolemizing,
this simplifies to:
seq_is_secure.2 :

  |-------
{1}   SecState(nth(seq, j!1)) IMPLIES SecState(nth(seq, j!1 + 1))

Rerunning step: (use "seq_transform")
Using lemma seq_transform,
this simplifies to:
```

```
seq_is_secure.2 :

{-1}  EXISTS x: nth(seq, j!1 + 1) = Transform(x, nth(seq, j!1))
  |-------
[1]   SecState(nth(seq, j!1)) IMPLIES SecState(nth(seq, j!1 + 1))

Rerunning step: (skolem!)
Skolemizing,
this simplifies to:
seq_is_secure.2 :

{-1}  nth(seq, j!1 + 1) = Transform(x!1, nth(seq, j!1))
  |-------
[1]   SecState(nth(seq, j!1)) IMPLIES SecState(nth(seq, j!1 + 1))

Rerunning step: (use "transition_state_secure")
Using lemma transition_state_secure,
this simplifies to:
seq_is_secure.2 :

{-1}  SecState(nth(seq, j!1)) => SecState(Transform(x!1, nth(seq, j!1)))
[-2]  nth(seq, j!1 + 1) = Transform(x!1, nth(seq, j!1))
  |-------
[1]   SecState(nth(seq, j!1)) IMPLIES SecState(nth(seq, j!1 + 1))

Rerunning step: (replace -2 (-1) rl)
Replacing using formula -2,
this simplifies to:
seq_is_secure.2 :

{-1}  SecState(nth(seq, j!1)) => SecState(nth(seq, j!1 + 1))
[-2]  nth(seq, j!1 + 1) = Transform(x!1, nth(seq, j!1))
  |-------
[1]   SecState(nth(seq, j!1)) IMPLIES SecState(nth(seq, j!1 + 1))

which is trivially true.

This completes the proof of seq_is_secure.2.

Q.E.D.



Run time  = 0.88 secs.
Real time = 1.39 secs.
```

GLOSSARY

	DEFINITION
Abstract data types	*An abstract data type (ADT) is characterized by the following properties: (1) It exports a type. (2) It exports a set of operations. This set is called interface. (3) Operations of the interface are the one and only access mechanism to the type's data structure. (4) Axioms and preconditions define the application domain of the type.*
BDDs(Binary Decision Diagrams)	*BDDs are a canonical and efficient way to represent and manipulate Boolean functions*
Dependent type	*A dependent type is a type consisting of several components where the type of (or the domain of) a later component depends on other earlier components*
Equational logic	*First-order equational logic consists of quantifier-free terms of ordinary first-order logic, with equality as the only predicate symbol*
Executable formal specification	*Executable formal specification can allow engineers to test (or simulate) the specified system on concrete data before the system is implemented*
First order predicate logic	*Predicate logic in which predicates take only individuals as arguments and quantifiers only bind individual variables.*
Higher order predicate logic	*Predicate logic in which predicates take other predicates as arguments and quantifiers bind predicate variables*
Inductive theorem prover	*Theorem prover with induction that is based on the explicit induction paradigm*
Inductively defined predicates	*Inductively defined predicates have several properties that can be used when formally proving theorems on the predicates: (1) The theorem that the predicate P satisfies the given rules (2) The induction theorem obtained from the fact that P is the least relation (3) The case analysis theorem on P*
Logical Framework	*The idea of a logical framework is to provide a single, universal formalism for representing formalisms so that an implementation of a formal system may be obtained by simply representing it in the framework*
Model Checker	*A technique used to detect errors or prove the absence in safety critical software and hardware systems; Based on the optimization of datastructure and algorithms*
Predicate subtypes	*A predicate subtype is a subtype defined by requiring the elements of the subtype to satisfy a predicate*

	DEFINITION
Proof checker	*A proof checker consists of "a system of language for mathematics" and "a program to check the proof written in such a language".*
Propositional logic	*Propositional Logic is concerned with propositions and their interrelationships. Roughly speaking, a proposition is a possible "condition" of the world about which we want to say something*
Quantifier free	*A formula is quantifier free if and only if it contains no quantifiers*
Recursively defined abstract data types	*Abstract data structures that are "tree-like" recursive data structures (i.e. lists and binary trees)*
Semantic subtype	*A semantic subtype is a subtype such that the set of the objects represented by the terms of the subtype is a subset of the objects represented by the terms of the parent type*
Subtyping	*If for every object o1: S there is o2:T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T.*
Syntactic subtype	*A syntactic subtype is a subtype such that the set of the syntactic elements (terms) of the subtype is a subset of the syntactic elements of the parent type*
Theorem prover	*A tool necessary to check and to manage proofs. It can come with decision procedures and other automated deduction which can make the proof much quicker and easier. The theorem prover is closely related to a proof checker since both are about "mechanical reasoning. The difference is whether the focus is on the computer checking the reasoning of the human, or on the human watching and guiding the computer's proof efforts. The more a system tries to be "smart", the more chance it has of being in the "theorem prover" category.*

REFERENCES

[And00] Andrews, Peter. *System Description: TPS: A Theorem Proving System for Type Theory*. <gtps.math.cmu.edu/tps-descr.ps>. 2000.

[And03] Andrews, Peter. *TPS Homepage*. <gtps.math.cmu.edu/tps-about.html>

[Arc01] Archer, Myla. *TAME: Using PVS Strategies for Special-Purpose Theorem Proving*. <chacs.nrl.navy.mil/publications/CHACS/2001/index2001.html>. 2001.

[Arg03] Argonne National Laboratory. *Otter: An Automated Deduction System: Homepage*. <www-unix.mcs.anl.gov/AR/otter/#accomp>

[Asc02] Ascent Technologies. <http://www.asc-tech.com/vv.htm>. January 2002.

[Bel73] Bell, D.E., and LaPadula, L.J., *Secure computer systems: mathematical foundations and model*. M74-244, The MITRE Corp., Bedford, Mass., May 1973.

[Bib77] Biba, K. *Integrity Considerations for Secure Computer Systems.* U.S. Air Force Electronic Systems Division Technical Report 760372, 1977.

[Car97] Cardelli, L., *Type Systems*, *Handbook of Computer Science and Engineering*, Chapter 103, CRC Press, 1997 <http://research.microsoft.com/Users/luca/Papers/TypeSystems.pdf>

[Cha02] Charpenter, Michel. *Tool Support for Formal Verification of Reactive Systems*. <www.cs.unh.edu/~charpov/ProofAssistant>. July 2002.

[Com03] Common Criteria Documentation. <http://www.commoncriteria.org/cc/cc.html>. 2003

[Cor03] Cornell University. *PRL Project "Proof/Program Refinement Logic": NuPrl Homepage*. <www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html>

[Dut97] Dutertre, B. and Stavridou, V., *Formal Requirements Analysis of an Avionics Control System.* IEEE transaction on Software Engineering.Vol. 23, NO.5, pg. 267 – 277, May 1997

[Far93] Farmer, W. *IMPS: An Interactive Mathematical Proof System*. <imps.mcmaster.ca/doc/imps-overview.pdf>. February 1993.

[Far95] Farmer, W. *IMPS*. <www-formal.Stanford.edu/clt/ARS/Entries/imps>. August 1995.

[Far03] Farmer, W. *IMPS Homepage*. <imps.mcmaster.ca>

[Fre02] Frey, P., Radhakrishnan, R. and Carter, H., *A Formal Specification and Verification Framework for Time Warp-Based Parallel Simulation*. IEEE Transaction on Software Engineering. Vol. 28, No. 1, January 2002

[Hei96] Heimdahl, M.P.E. and Czerny, B.J., *Using PVS to analyze hierarchical state-based requirements for completeness and consistency.* High-Assurance Systems Engineering Workshop, IEEE. pgs. 252 – 262, October 1996

[Ibm03] IBM Vienna Laboratory. *Vienna Development Method Homepage*. <www.ifad.dk/vdm/vdm.html>

[Inr03] Inria, *Coq Homepage*. <coq.inria.fr>

[Irv03] Irvine, C., *Security Policy Modeling*. Center for INFOSEC Studies and Research. 2003

[Iyo03] Iyoda, J., *HOL Homepage*. <www.cl.cam.ac.uk/Research/HVG/HOL>

[Kau02] Kaufamm, M., Manolios, P., and Moore, J.S., *ACL2 – ComputerAided Reasoning: An Approach*. August 2002

[Kav02] Kavanagh, K., *The Role of Mathematical Logic in Designing Secure Systems*. 2002

[Kes03] Kestrel Institute. *SpecWare Homepage*. <www.kestrel.edu/HTML/prototypes/specware.html>

[Koh99] Kohlhase, M., *Database of Existing Mechanized Reasoning Systems*. <http://www-formal.stanford.edu/clt/ARS/systems.html>. June 1999

[Kro02] Kroening Daniel. *Application Specific Higher Order Logic Theorem Proving.*
< www.kroening.com/papers/verify2002.pdf>. 2002.

[Law02] Lawford, M. and Hu, X., *Right on Time: Pre-verified Software Components for Construction of Real-Time Systems*. Nov. 2002.

[Lam74] Lampson, B., *Protection. ACM Operating Systems Review*, 8(1):18-24, January 1974.

[Man99] Manna, Zohar. *An Update on Step: Deductive-Algorithmic Verification of Reactive Systems*. <www.stanford.edu/~finkbein/publications/tools98/tools99.pdf>.1999.

[Mer96] Merriam, N.A. and Harrison, M.D., *Evaluating the Interfaces of Three Theorem Proving Assistants*. 1996

[Moo03] Moore and Kaufmann. *ACL2 Version 2.7 Hompage*. <www.cs.utexas.edu/users/moore/acl2/acl2-doc.html>

[Nog02] Nogin, Aleksey. *A Review of Theorem Provers*. <www.cs.cornell.edu/Nuprl/PRLSeminar/PRLSeminar01_02/Nogin/PRLseminar7b.pdf>. February 2002.

[Ora03] ORA Canada. *Z/Eves Homepage*.<www.ora.on.ca/z-eves/welcome.html>

[Owr99] Owre, S. and Shankar, N., *The Formal Semantics of PVS.* <http://www.csl.sri.com/papers/csl-97-2/>. March 1999

[Owr01] Owre, S. and others. *PVS Language Reference.* SRI International. December 2001

[Owr01*] Owre, S. and others. PVS System Guide. SRI International. December 2001

[Pai01] Paige, R.F. and Ostroff, J.S., *Metamodelling and Conformance Checking with PVS*. 2001

[Pau02] Paulson, Lawrence. *Introduction to Isabelle*. <www.cl.cam.ac.uk.Research/HVG/Isabelle/dist/Isabelle2002/doc/intro.pdf>. March 2002.

[Pau03] Paulson, L. and Nipkow, T., *Isabelle Homepage*. <www.cl.cam.ac.uk/Research/HVG/Isabelle>

[Pfe03] Pfenning and Schurmann. *The Twelf Project*. <www.cs.cmu.edu/~twelf>

[Rad87] Radium. *FTLS Must Accurately Describe TCB Operations (C1-CI-01-87).* <http://www.radium.ncsc.mil/tpep/library/interps/0237.html>. 1987

[Rus96] Rushby, J.M. *PVS:Combining Specification, Proof Checking.* <people.cs.uchicago.edu/~robby/contract-reading-list/pvs-cav96.ps>.1996.

[Rus98] Rushby, J., Owre, S. and Shankar, N., *Subtypes for Specifications: Predicate Subtyping in PVS*. IEEE Transaction on Software Engineering. Vol. 24, No. 9, September 1998

[Sec98] Secure Computing Corporation. *Composability for Secure Systems Top Level Specification (TLS).* <http://www.securecomputing.com/css/tls.pdf>. July 1998

[Sec96] Secure Computing Corporation. *DTOS Formal Security Policy Model(FSPM).* pgs. 1-20, September 1996.

[Sec96*] Secure Computing Corporation. *DTOS Formal Top-Level Specification (FTLS).* pgs. 1-20, December 1996

[Sha01] Shankar, N. and others. *PVS Prover Guide*. SRI International. November 2001. <http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf>

[Sri03] SRI International. *PVS Homepage*. <pvs.csl.sri.com>

[Sta03] Stanford University. *STeP Homepage*. <www-step.stanford.edu>

[Sul03] Sullivan, E.C., *Security Policy Models: The Bell-Lapadula Model.*

[Tum03] TU Muenchen. *AutoFOCUS Homepage*. <autofocus.informatik.tu-muenchen.de/index-e.html>

[Uom98] University of Manchestor. *VDM-Vienna Development Method.* <www.vienna.cc.evdm.htm>. 1998.

[Wil97] Wilding, M.M. *Robust Computer System Proofs in PVS.* <citeseer.nj.nec.com/wilding97robust.html>. 1997

[You96] Young, William. *The Specification of a Simple Autopilot in ACL2.* <www.cl.cam.ac.uk/~mjcg/BDD/facs21-talk.ps.gz>. July 1996.

[You96*] Young William, *Comparing Verification System: Interactive Consistency in ACL2.* <www.cs.utexas.edu/users/moore/publications/others/interactive-consistency-young.ps>. 1996

[Zha98] Zhang, Wenhui. *Evaluation of verification tools* <
www.ifi.uio.np/~adapt/adapt-ft-05.ps.gz>. 1998

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Dr. Ernest McDuffie
   National Science Foundation
   Arlington, VA

4. RADM Zelebor
   N6/Deputy DON CIO
   Arlington, VA

5. Russell Jones
   N641
   Arlington, VA

6. David Wirth
   N641
   Arlington, VA

7. CAPT Sheila McCoy
   Headquarters U.S. Navy
   Arlington, VA

8. CAPT Robert Zellmann
   CNO Staff N614
   Arlington, VA

9. Dr. Ralph Wachter
   ONR
   Arlington, VA

10. Dr. Frank Deckelman
    ONR
    Arlington, VA

11. Richard Hale
    DISA
    Falls Church, VA

12. George Bieber
OSD
Washington, DC

13. Deborah Cooper
DC Associates, LLC
Roslyn, VA

14. David Ladd
Microsoft Corporation
Redmond, WA

15. Marshall Potter
Federal Aviation Administration
Washington, DC

16. Ernest Lucier
Federal Aviation Administration
Washington, DC

17. Keith Schwalm
DHS
Washington, DC

18. RADM Joseph Burns
Fort George Meade, MD

19. Douglas Maugham
DARPA
Arlington, VA

20. Howard Andrews
CFFC
Norfolk, VA

21. Steve LaFountain
NSA
Fort Meade, MD

22. Penny Lehtola
NSA
Fort Meade, MD

23. Dr. George Dinolt
Naval Postgraduate School
Monterey, CA

24. Professor Timothy Levin
Naval Postgraduate School
Monterey, CA

25. Sonali Ubhayakar
Naval Postgraduate School
Monterey, CA